Array of Objects

```
/* Array of Objects
datatype arr_name[] = new type[size];
Student s[ ] = new Student[10];
datatype[] arr_name = new type[size];
Student[] s = new Student[10]; */
import java.util.*;
class Student
 {
   int id ;
        String name;
        double avg;
        Student(int id)
        {
                this.id = id;
        }
        Student(int id , String name )
        {
                this(id);
                this.name= name;
        }
         Student (int id , String name , double avg)
        {
                this(id,name);
                this.avg = avg;
        }
      void display( )
        {
          System.out.println(id + " , " + name + ", " + avg);
        }
 }
```

```
public class Filter
{
        public static void main(String args[])
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the number of Students");
                int n = sc.nextInt();
                Student s[] = new Student[n];

                for(int i=0; i<s.length ; i++)
                {
System.out.println("Enter the Student Details of -> " +(i+1));
                        int x = sc.nextInt();
                        String str = sc.next();
                        double y = sc.nextDouble();
                        s[i] = new Student(x,str,y);
                }

 System.out.println("Enter the Average to display list ");
                double p = sc.nextDouble();

                for(int i=0; i< n ; i++)
                  if(s[i].avg >= p)
                                s[i].display();
        }
}
```

StringTokenizer class

StringTokenizer Class - java.util.*; the processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts or tokens.
The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner.

The StringTokenizer constructors

StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean flag)

The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.

If flag is true, then the delimiter is also returned as token.

The methods define in StringTokenizer are

Method Name	Description
int countTokens( )	The number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns true if one or more tokens remain in the string
boolean hasMoreTokens( )	Returns true if one or more tokens remain in the string
Object nextElement( )	Returns the next token as an Object.
String nextToken( )	Returns the next token as a String.

```
//StringTokenizer Exampl-1
import java.util.*;
public class  TokenDemo
{
  public static void main(String[] args)
       {
              Scanner sc = new Scanner(System.in);
              System.out.println("Enter a line of Text");
               String str = sc.nextLine();

               StringTokenizer st=new StringTokenizer(str);
System.out.println("The Count of Tokens is " +st.countTokens()); //3
              System.out.println("The tokens are");
                    while(st.hasMoreTokens())
                          {
                                String s = st.nextToken();
                                System.out.println(s);
                          }
System.out.println("The Count of Tokens is " +st.countTokens()); // 0
System.out.println("Enter  a string with # as delimeter");

String str1 = sc.nextLine();

StringTokenizer st1 = new StringTokenizer(str1,"#",true)
System.out.println("The Count of Tokens is " +st1.countTokens()); //5
                     while(st1.hasMoreTokens())
                                {
                                  String s = st1.nextToken();
                                 System.out.println(s);
                          }

                    }
              }
```

//StringTokenizer Exampl-2

```
import java.util.*;
public class  SumDemo
{
	public static void main(String[] args)
	{
	 Scanner sc = new Scanner(System.in);
	 System.out.println("Enter a line of Integers");
	 String str = sc.nextLine();

	  StringTokenizer st=new StringTokenizer(str);
System.out.println("The Count of Tokens is " +st.countTokens());
	int n,sum=0;
	System.out.println("The tokens are");
	while(st.hasMoreTokens())
		{
			String s = st.nextToken();
			n=Integer.parseInt(s);
			System.out.println(n);
			sum=sum+n;
		}

	System.out.println("Sum of Tokens is :"+sum);
System.out.println("The Count of Tokens is " +st.countTokens());// 0
	}
}
```

1. Evolution of Java

What is Java?

Java is a programming language and a platform.

- **Java is a high level, robust, secured and object-oriented programming language.**

- **Platform: Any hardware or software environment in which a program runs is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.**

Where it is used?

1. **Desktop Applications such as acrobat reader, media player, antivirus etc.(Window Applications)**
2. **Web Applications such as irctc.co.in, cleartrip.com etc.**
3. **Enterprise Applications such as banking,e-commerce applications.**
4. **Mobile Applications (J2ME)**
5. **Embedded System**
6. **Smart Card**
7. **Robotics**
8. **Games etc.**

Types of Java Applications

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications. (J2SE)

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.(J2SE)

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications. (J2EE)

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.(J2ME)

Evolution of Java

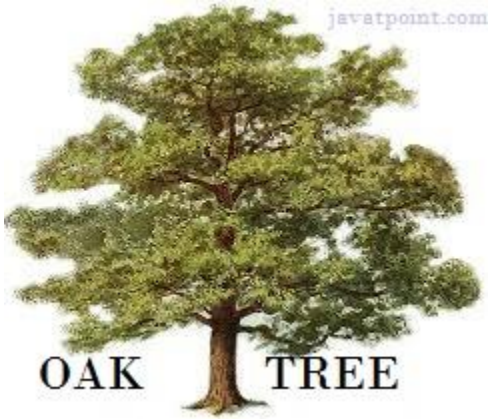
Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.



Why sun choosed "Oak" name?

5) Why Oak? Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.

6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Why sun choose "Java" name?

7) Why they choosed java name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name not an acronym.

10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called Java one of the Ten Best Products of 1995.

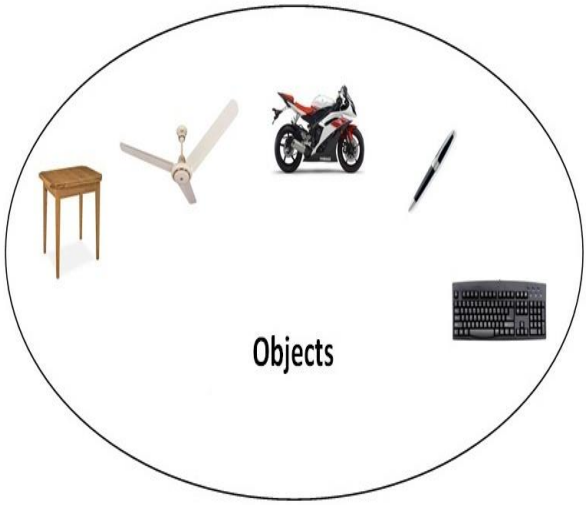
12) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that have been released. Current stable release of Java is Java SE 8.

1. **JDK Alpha and Beta (1995)**
2. **JDK 1.0 (23rd Jan, 1996)**
3. **JDK 1.1 (19th Feb, 1997)**
4. **J2SE 1.2 (8th Dec, 1998)**
5. **J2SE 1.3 (8th May, 2000)**
6. **J2SE 1.4 (6th Feb, 2002)**
7. **J2SE 5.0 (30th Sep, 2004)**
8. **Java SE 6 (11th Dec, 2006)**
9. **Java SE 7 (28th July, 2011)**
10. **Java SE 8 (18th March, 2014)**
11. **Java SE 9 (September, 21st 2017)**
12. **Java SE 10(March, 20th 2018)**
13. **Java SE 11 (18th March, 2014)**
14. **Java SE 12(March, 19th 2019)**
15. **Java SE 13(September, 17th 2019)**
16. **Java SE 14(March, 17th 2020)**
17. **Java SE 15(Expected in September 2020)**

2. OOPs Concepts / OOPs Principles



Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- **Object**
- **Class**
- **Inheritance - OOP Principle -2**
- **Polymorphism - OOP Principle -3**
- **Abstraction**
- **Encapsulation - - OOP Principle -1**

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class: Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Property transfer from Grand Parent to Parent – to children.

This is one way from top to bottom of hierarchy.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

i.e The process of representing one form in multiple forms is known as polymorphism.

e.g. When we are in class – student ,
When we are in Market – Customer,
When we are in Home – Son / Daughter.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw,

dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Exposing only the required essentials characteristics & behavior with respect to the context.

Shows important things to the user and hide the internals details.

e.g. ATM , BIKE

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation.

Date Hiding + Abstraction

For example: capsule, it is wrapped with different medicines, Power Steering Car (Internally lot of Components tightly Coupled together)

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Capsule

3. BASIC STRUCTURE OF A JAVA PROGRAM

```
public class Simple
{
   public static void main(String args[])
    {
            System.out.println("Welcome To the World of Java");
    }
}
```

save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument.
- **System.out.println()** is used print statement.

Basic Structure of Java Program

//Approach -1

```
public class Simple
{
   public static void main(String args[])
    {
            System.out.println("Welcome To Smart Learning of Java");
    }
}
```

File Name : Simple.java
Compile : javac Simple.java
Execute/Run : java Simple

//Approach -2

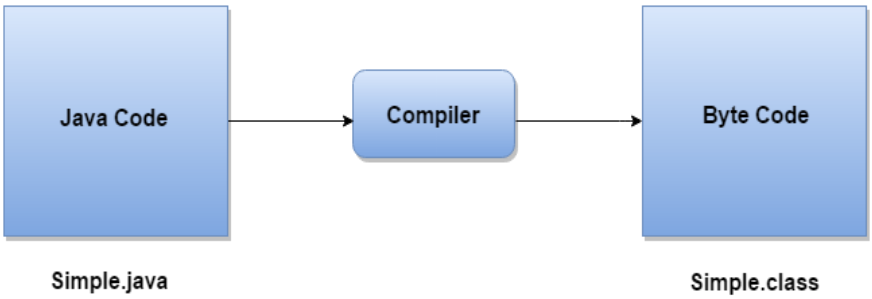
```
class Simple
{
   public static void main(String args[])
    {
            System.out.println("Welcome To Smart Learning of Java");
    }
}
```

File Name : demo.java
Compile : javac demo.java
Execute/Run : java Simple

4. Java Byte Code (Java Magic)

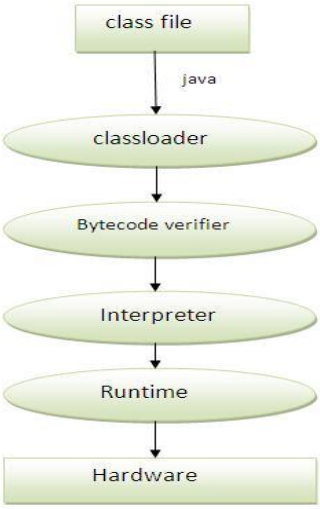
What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into byte code.



What happens at runtime?

At runtime, following steps are performed:



Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

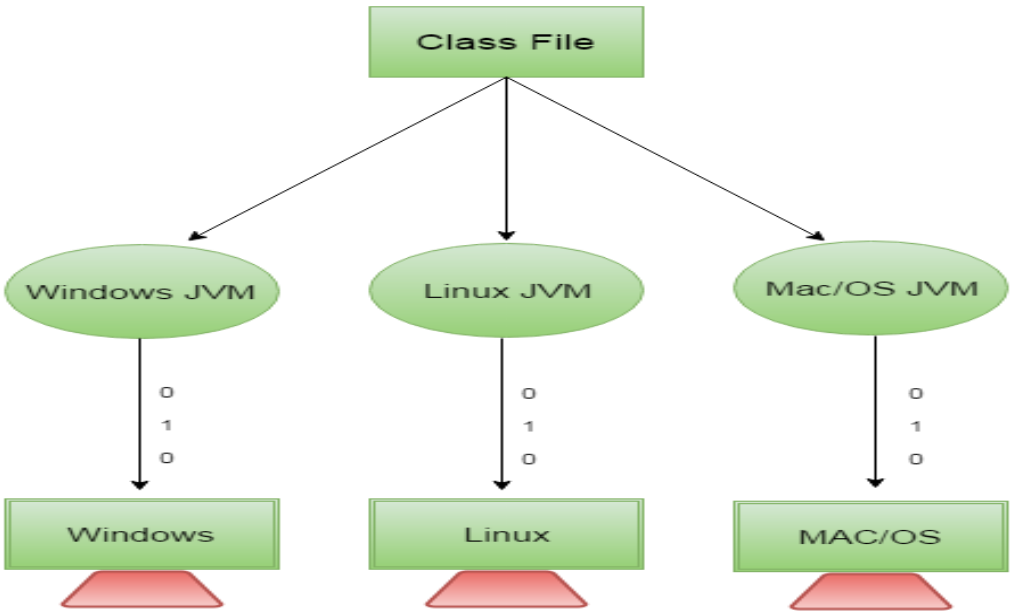
Interpreter: read byte code stream then execute the instructions.

Byte Code:

- **The output of a Java compiler is not executable code. Rather, it is byte code.**

- ***Byte code* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*.**
- **In essence, the original JVM was designed as an *interpreter for byte code*.**

- **Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.**



- **Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java byte code.**

- **The execution of byte code by the JVM is the easiest way to create truly portable programs.**

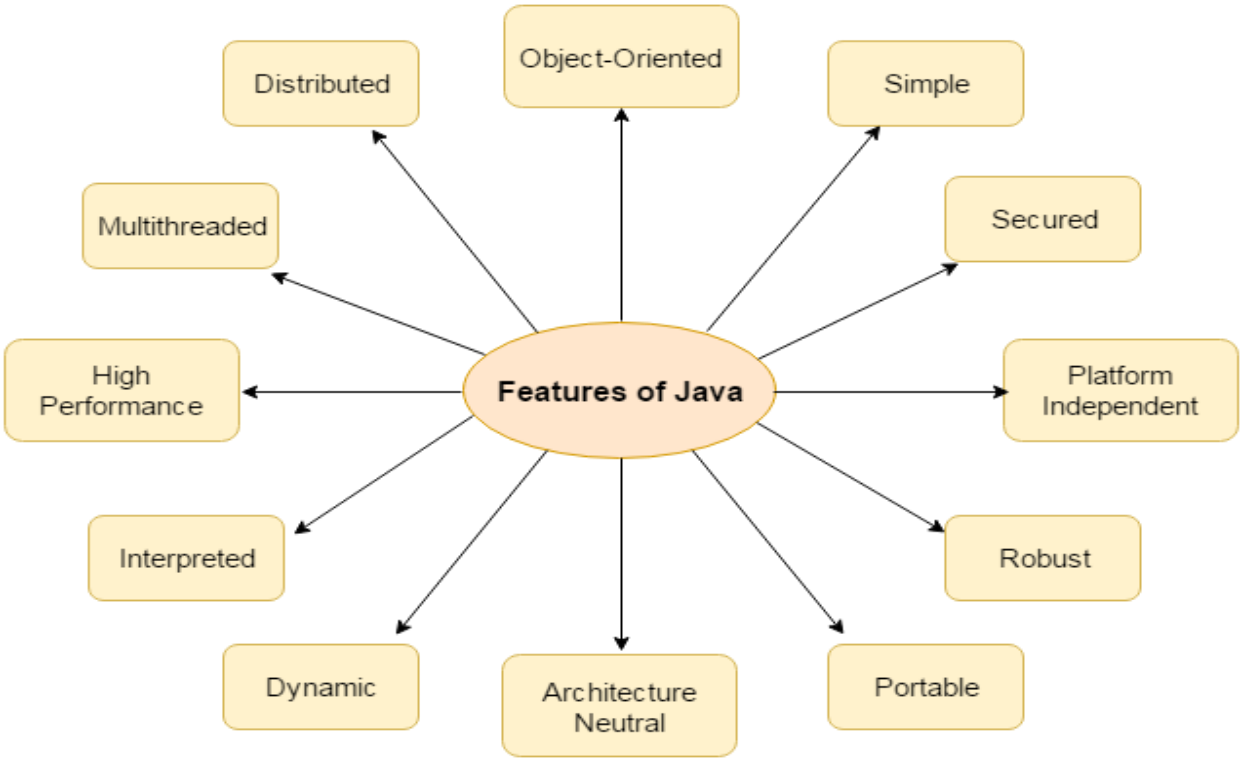
- **The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.**

- **In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code.**

- **To improve performance JIT Compiler is used (Just In Time Compiler). JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation and during execution.**

5. Features of Java / Java Buzz Words

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.



1. **Simple**
2. **Object-Oriented**
3. **Portable**
4. **Platform independent**
5. **Secured**
6. **Robust**
7. **Architecture neutral**
8. **Dynamic**
9. **Interpreted**
10. **High Performance & Multithreaded**
11. **Distributed**

Simple

According to Sun, Java language is simple because:

Syntax is based on C++ (so easier for programmers to learn it after C++).

removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

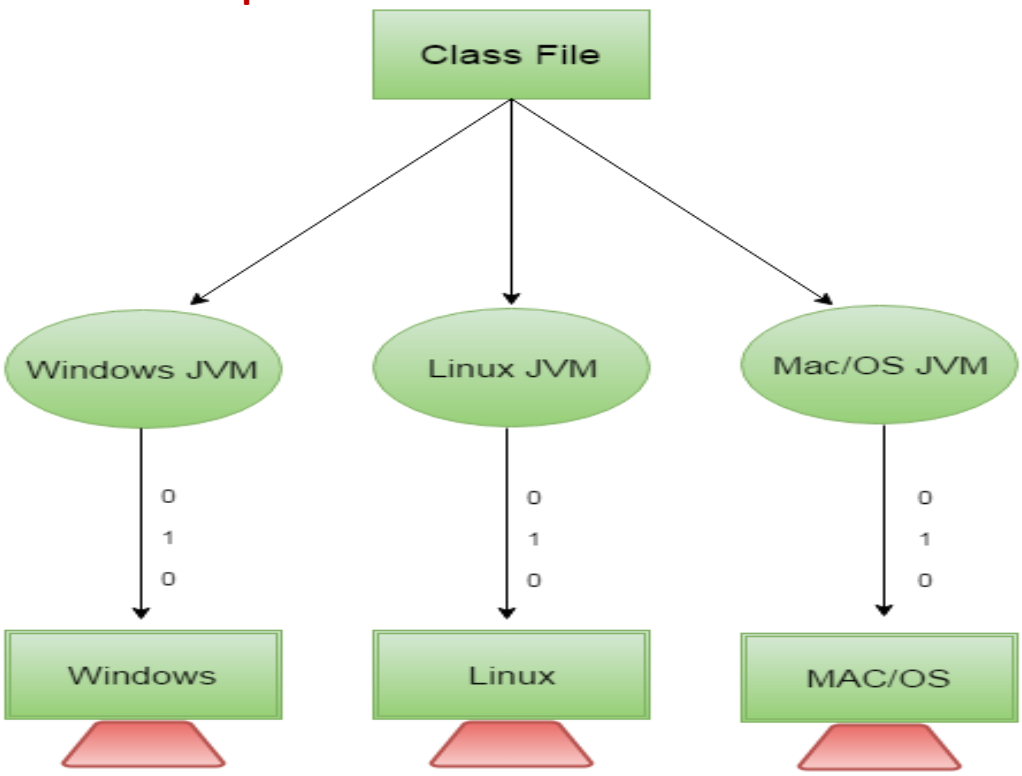
Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. **Object**
2. **Class**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**
6. **Encapsulation**

Platform Independent



A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

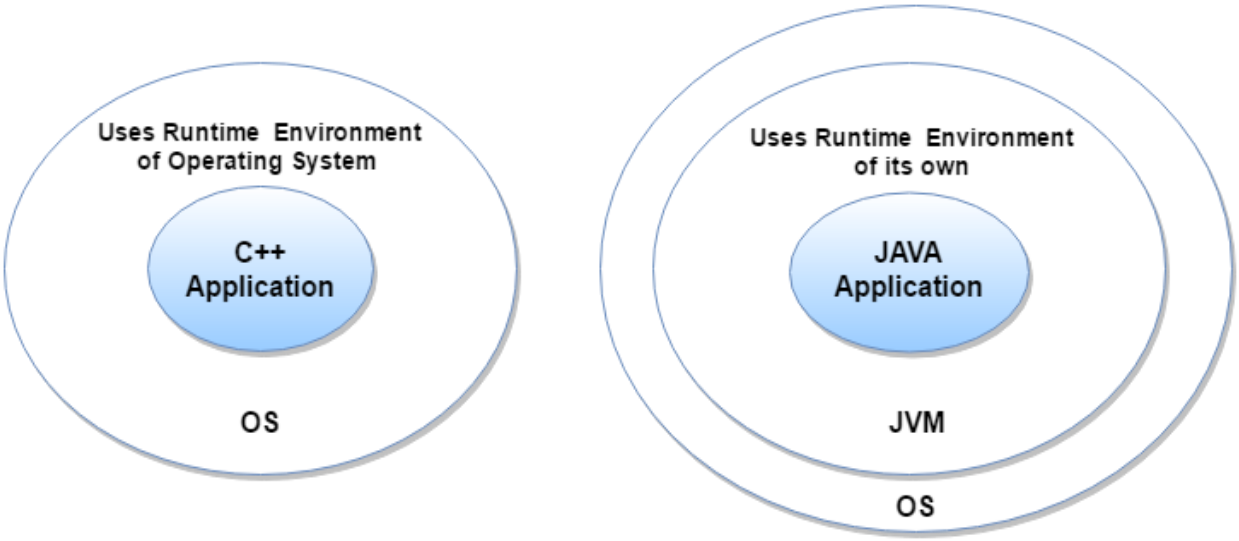
1. **Runtime Environment**
2. **API(Application Programming Interface)**

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is secured because:

- **No explicit pointer**
- **Java Programs run inside virtual machine sandbox**



- **Classloader: adds security by separating the package for the classes of the local file system from those that are imported from network sources.**
- **Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.**
- **Security Manager: determines what resources a class can access such as reading and writing to the local disk.**

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is

automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

There are no implementation dependent features e.g. size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable

We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

8. Data Types in Java

"Data types represent the different values to be stored in variable"

Data type specifies

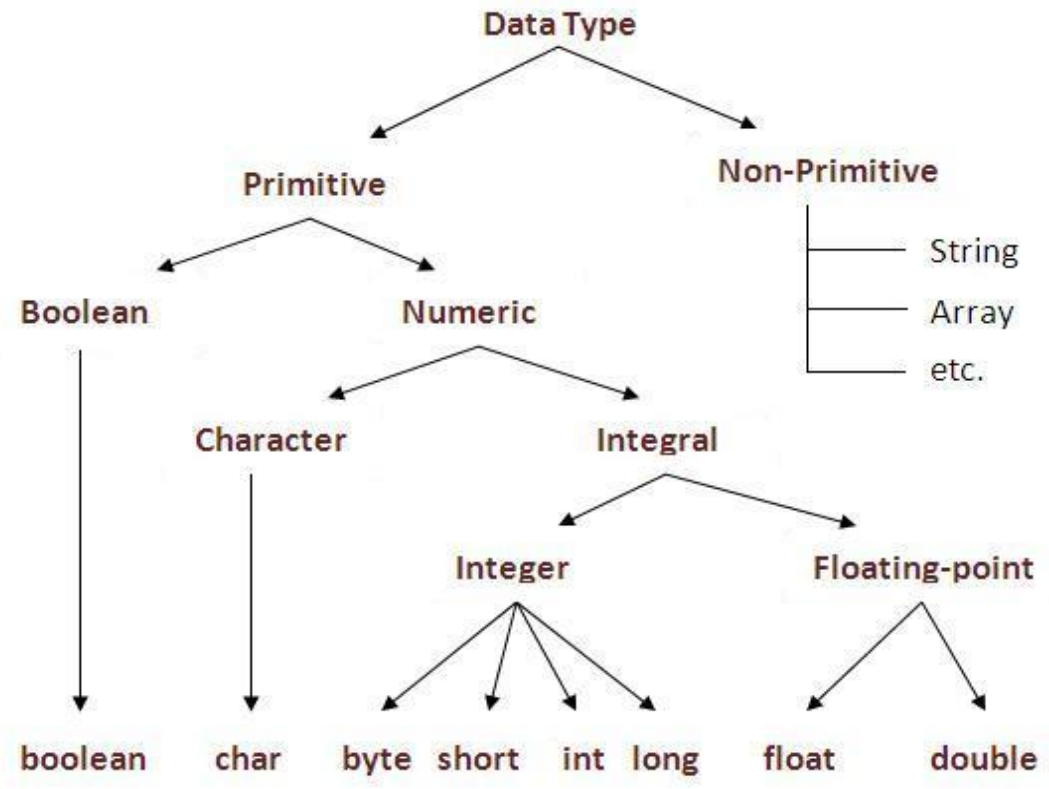
1. **Type of Value stored in a variable**
2. **Range of Values to a variable**
3. **Size of variable**
4. **Operations Performed**

Java is a strongly typed language

1. **Every variable has a type, every expression has a type, and every type is strictly defined.**
2. **All assignments, whether explicit or via parameter passing in method calls, are checked for Type compatibility.**
3. **There are no automatic coercions or conversions of conflicting types as in some languages.**
4. **The Java compiler checks all expressions and parameters to ensure that the types are compatible.**
5. **Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.**

In java, there are two types of data types:

1. **Primitive data types**
2. **Non – Primitive Data Types**



Data Type	Default Value	Default size (Width)
boolean	False	1 bit (0 or 1)
char	'\u0000'	2 byte (16 bits)
byte	0	1 byte (8 bits)
short	0	2 byte (16 bits)
int	0	4 byte (32 bits)

long	0L	8 byte (64bits)
float	0.0f	4 byte (32 bits)
double	0.0d	8 byte (64 bits)

- All of integer data types are signed, positive and negative values. Uses 2's complement notation for representing integer values.
- Java does not support unsigned, positive-only integers
- Java implements the standard IEEE–754 for floating-point types.
- There are two kinds of floating-point types, float and double which represent single precision (32bits) - and double precision (64bits) - numbers.
- Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false.

Integer Type Range

Name	Width in Bits	Range
long	64 (8 Bytes)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32 (4 Bytes)	–2,147,483,648 to 2,147,483,647
short	16 (2 Bytes)	–32,768 to 32,767
byte	8 (1 Byte)	–128 to 127

Floating Point Range

Name	Width in Bits	Range
double	64 (8 Bytes)	4.9e–324 to 1.8e+308
float	32 (4 Bytes)	1.4e–045 to 3.4e+038

Character type Range

Name	Width in Bits	Range
char	16 (2 Bytes)	0 to 65,535 (no negative characters) UNI – CODE system

Java Character System

Java Character set uses Unicode system.

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Before Unicode, there were many language standards:

ASCII (American Standard Code for Information Interchange) for the United States.

In Unicode, character holds 2 byte, so java also uses 2 byte for characters.

- **Java uses *Unicode* to represent characters.**
- **Unicode defines a fully international character set that can represent all of the characters found in all human languages.**
- **It is a unification of dozens of character sets, such as Latin, Greek, Arabic,**
- **Cyrillic, Hebrew, Katakana, Hangul, and many more.**
- **The range of a char is 0 to 65,535.**
- **There are no negative chars.**
- **The standard set of characters known as ASCII still ranges from 0 to 127 as always,**
- **Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.**

2. Programs to Read Data from Standard Input

Using Java Scanner Class to read input

Scanner is a class in "java.util" package used for obtaining the input of the primitive types like int, double, and strings etc. It is the easiest way to read input in a Java program

The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values. Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Commonly used methods of Scanner class

Method	Description
public String next()	It returns the next token (string) from the scanner.
public String nextLine()	It moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	It scans the next token as a byte.
public short nextShort()	It scans the next token as a short value.
public int nextInt()	It scans the next token as an int value.
public long nextLong()	It scans the next token as a long value.
public float nextFloat()	It scans the next token as a float value.
public double nextDouble()	It scans the next token as a double value.
public double nextChar()	It scans the next token as a charcater value.

// Example to read and display data

```
import java.util.*;
import java.io.*;
public class DataRead
    {
        public static void main(String args[])
          {
            Scanner sc=new Scanner(System.in);

            System.out.println("Enter your rollno");
            int rollno=sc.nextInt();

            System.out.println("Enter your name");
            String name=sc.next();

            System.out.println("Enter your fee");
            double fee=sc.nextDouble();

           System.out.println(“Roll No  : “ +rollno);
           System.out.println(“Name is “ +name);
           System.out.println(“Feee is “ +fee);

           //System.out.println("Rollno:"+rollno  +" name:" +name + " fee:" +fee);

           sc.close();

        }
    }
```

Example Programs:

//Program to read a number and check it is even or odd.

```
import java.util.*;
public class Even
{
        public static void main(String[] args)
        {
           Scanner sc = new Scanner(System.in);
           System.out.println("Entera a number");
           int n = sc.nextInt();

           if( n % 2 == 0)
                System.out.println("Even Number " +n);
           else
                    System.out.println("Odd Number "+n);

        }
}

//Program to find the average of a student in three subjects marks (Double).

import java.util.*;

public class Grade
{
        public static void main(String[] args)
        {
           Scanner sc = new Scanner(System.in);
           System.out.println("Entera the Marks in 3 Subjects");
           double m1 = sc.nextDouble();
           double m2 = sc.nextDouble();
           double m3 = sc.nextDouble();

           double avg = (m1+m2+m3)/3;
           System.out.println("The average is "+avg);

        }
}
```

9. Control Statements

1. Conditional / Selection Statements

Java If-else Statement

The Java *if statement* is used to test the condition. It checks Boolean condition: *true* or *false*. There are various types of selection statement in java.

- **if statement (one way-selection)**

```
if(condition)
  {
     //code to be executed
  }
```

- **if-else statement (two way-selection)**

```
if(condition)
  {
     //code to be executed
  }
else
 {
     //code to be executed
 }
```

- **nested if statement (if inside another if)**

```
if(condition1)
  {
          if(condition2)
        {
          //code to be executed
        }

       //code to be executed
  }
```

- **if-else-if ladder (Multi way selection)**

```
if(condition1)
  {
     //code to be executed
  }
else if(condition2)
{
     //code to be executed
}
----------------------
else
{
     //code to be executed
}
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

```
switch(expression)
  {
     case value1:  //code to be executed;
                      break;  //optional
     case value2:  //code to be executed;
                      break;  //optional
      ......
     default:
                  // code to be executed if all cases are not matched;
  }
```

***case values can be integer constant, character constant and string constant.

```
public class Example
        {
        public static void main(String[] args)
        {
           int number=20;
           switch(number)
           {
            case 10: System.out.println("CSE");break;
            case 20: System.out.println("EEE");break;
            case 30: System.out.println("ECE");break;
            default:System.out.println("Invalid input");
           }
        }
}
```

Java Switch Statement is fall-through

The java switch statement is fall-through. It means it executes all statement after first match if break statement is not used with switch cases.

```
    public class Example2

        {
          public static void main(String[] args)
             {
              int number=20;
              switch(number)
                 {
                   case 10: System.out.println("10");
                   case 20: System.out.println("20");
                   case 30: System.out.println("30");
                   default:System.out.println("Hello");
                 }
             }
        }
```

2. Loop / Repetitive / Iterative Statements

Java while Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

***** It is an event based loop, entry controlled loop.**

Syntax:

```
while(condition)
  {
    //code to be executed
  }
```

Java Infinitive While Loop

If you pass true in the while loop, it will be infinitive while loop.

```
while(true)
  {
    System.out.println("never ends");
  }
```

Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

*** It is an event based loop, exit controlled loop

Syntax:

```
do
 {
   //code to be executed
  }while(condition);
```

If you pass true in the do-while loop, it will be infinitive do-while loop.

```
   do{
      System.out.println("Never Ends");
   }while(true);
```

Java for Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

*** It is a counter controlled loop, entry controlled loop

There are three types of for loop in java.

- **Simple For Loop**
- **for-each or Enhanced For Loop**

1. Java Simple for Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

```
for(initialization;condition;incr/decr)
        {
         //code to be executed
        }
```

Example:
```
for(int i=1;i<=10;i++)
{
   System.out.println(i);
}
```

2. Java for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```
for(type var: array)
  {
    //code to be executed
   }
```

```
int list[] = { 11, 2, 3, 14, 5, 62, 7, 8, 9, 10 };
int sum = 0;
    for(int    x :   list)
      {
         System.out.println("Value is: " + x);
         sum = sum + x;
       }
```

Java Infinitive for Loop

```
for( ; ; )
   {
    System.out.println("Never Ends");
   }
```

3. Jump Statements

Java break Statement

- The Java ***break*** is used to break loop or switch statement. It breaks the current flow of the program at specified condition.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- In case of inner loop, it breaks only inner loop.

```
for(int i=1;i<=10;i++)
  {
      if(i==5)
       {
          break;
       }
   }
  System.out.println(i);
```

Java Continue Statement

- The Java ***continue statement*** is used to continue loop.
- We might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- It continues the current flow of the program and skips the remaining code at specified condition.
- In case of inner loop, it continues only inner loop.

```
for(int i=1;i<=6;i++)
  {
     if(i==3)
    {
     continue;
   }
   System.out.println(i);
 }
```

Java return Statement

- The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- At any time in a method the return statement can be used to cause execution to branch back to the caller of the method.
- Thus, the return statement immediately terminates the method in which it is executed.

```
public class Demo
{
        public static void main(String[] args)
        {
                int res;
                res = add(20,30);
                System.out.println("The result is "+res);
        }

        public static int add(int a , int b)
         {
            return(a+b);
         }
}
```

10. Programs to Demonstrate Control Statements

[Hands on Session]

[Read Data From input Source]

1. **Program to find net salary of an employee considering basic pay as <=20000 and above.**
2. **Program to find the grade of a student using switch.**
3. **Program to find the sum of the digits of a number.**
4. **Program to display a number in name.**
   [123 -> one two three]
5. **Program to find factorial of a number using do-while.**
6. **Program to check a number is perfect number or not.**
7. **Program to display a list of fruits in an array using for-each loop.**
8. **Program to demonstrate use of break statement.**
   [Eg: Prime number or not.]
9. **Program to demonstrate use of continue statement.**
   [Eg: the numbers divisible by 3 from 1 to 30.]

11. JVM Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java byte code can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

JVM is

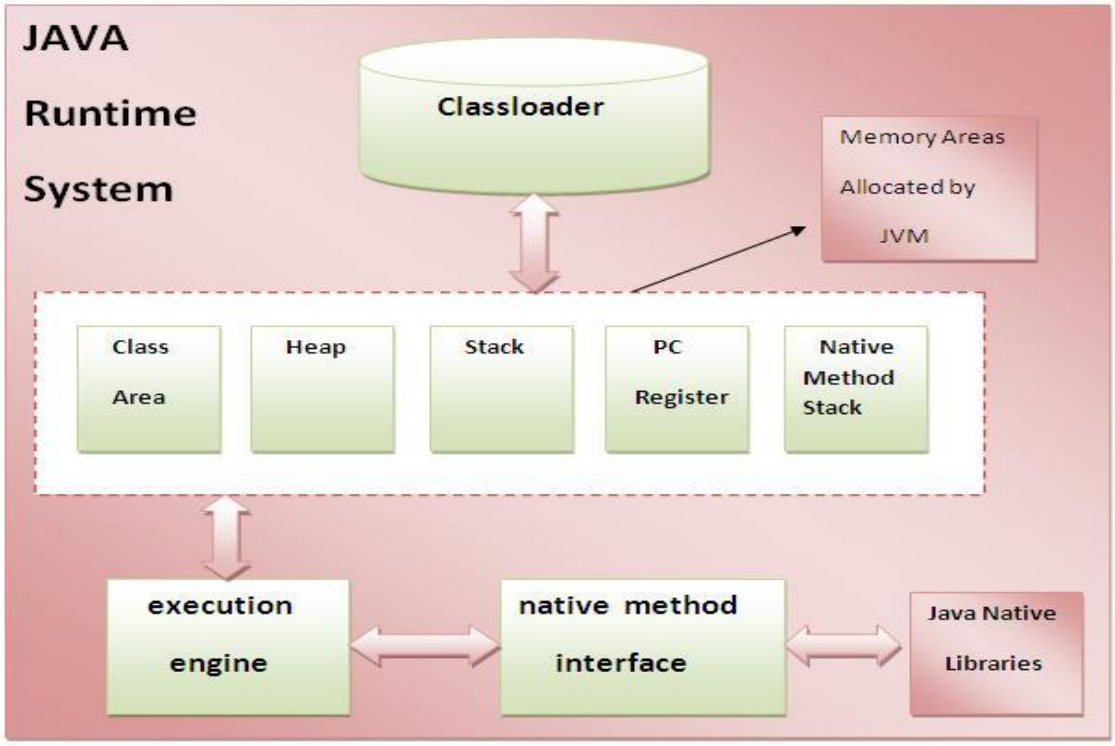
1. **A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.**
2. **An implementation known as JRE (Java Runtime Environment).**
3. **Runtime Instance, Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.**

The JVM performs following operation:

- **Loads code**
- **Verifies code**
- **Executes code**
- **Provides runtime environment**

JVM provides definitions for the:

- **Memory area**
- **Class file format**
- **Register set**
- **Garbage-collected heap**
- **Fatal error reporting etc.**



1) Classloader

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, class data , field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

a) A virtual processor

b) Interpreter: Read byte code stream then execute the instructions.

c) Just-In-Time (JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term compiler Refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

4. Garbage Collector: Automatic freeing of Heap Memory.

JRE

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs is also actively released by other companies besides Sun Micro Systems.

JRE

JDK

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.

JDK

12. Java Variables

Variable is a name of memory location.

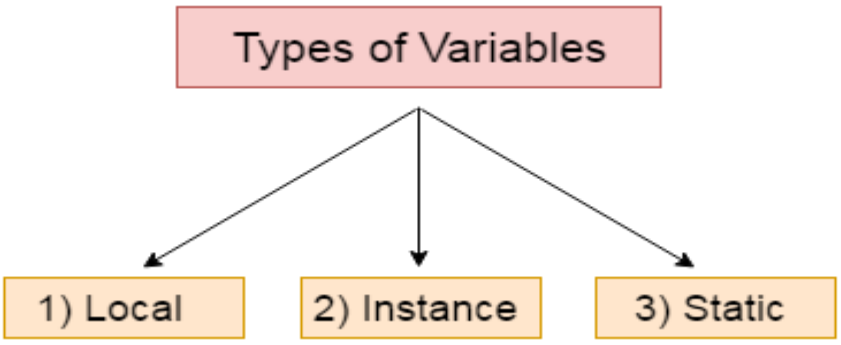
A Variable Has

1. **NAME (VALID IDENTIFIER)**
2. **VALUE**
3. **TYPE**
4. **SCOPE & LIFETIME**

```
datatype  var_name = [initial value]
int data=50;          //Here data is variable
int x,y;
float p=3.142, result;
char grade =’A’;
```

Types of Variable

There are three types of variables in java:



1) Local Variable

A variable which is declared inside the method is called local variable.
(Stored in Stack Area of Memory)

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable. This is not declared as static. (Stored in Heap area of Memory).

3) Static variable

A variable that is declared as static is called static variable. It cannot be local. It belongs to a class. (Stored in Class Area of Memory).

4) Reference variable

A variable that refers to object of a class. (Stored in Stack Area of Memory)

```
Scanner sc = new Scanner(System.in); //sc-Reference variable-stack
```

```
class A
{
        int data=50;//instance variable-heap
        static int m=100;//static variable-class area

        void display( )
        {
                int n=90;//local variable- stack area
                 ----------
                 ----------
         }
}
```

Dynamic Initialization

Java allows variables can be initialized dynamically using any expression valid at the time the variable is declared.

```
class  DynInit
      {
           public static void main(String args[])
             {
                  double a = 3.0, b = 4.0;
                  a=a+5;
                  b=b-1;
                  double c = Math.sqrt(a * a + b * b);
                 System.out.println("Result is  " + c);
              }
      }

for( int i = 0;i<10;i++)
  {

  }
```

But in C we Write it as

```
for(i = 0;i<10;i++)
  {

  }
```

"The scope and Life time of variables in java is local to the block in which they defined." (Except for static variables).

Java Variable Example:

```
public class Simple
{
    public static void main(String[] args)
        {
            int a=10;
            int b=10;
            int c=a+b;
            System.out.println(“The sum is “ +c);
        }
}
```

C - Variable Example:

```
void main( )
{
          int a=10;
          int b=10;
          int c;
          c=a+b-2;
          System.out.println(“The sum is “ +c);

}
```

13. Operators in Java

a) Arithmetic Operators (Arithmetic Operation)

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus (Applied to both integer and float data)
++	Increment
+=	Addition assignment (x=x+1 as x += 1)
– =	Subtraction assignment (x=x-1 as x -=1)
*=	Multiplication assignment (x=x*2 as x *= 1)
/=	Division assignment (x=x/2 as x /=1)
%=	Modulus assignment (x=x%4 as x %=1)
– –	Decrement

```
public class OperatorDemo
 {
     public static void main(String args[])
         {
                int a=10,b=5;
                System.out.println(a+b); //15
                System.out.println(a-b); //5
                System.out.println(a*b); //50
                System.out.println(a/b); //2
                System.ot.println(a%3); //1
                 double c=42.25;
                 System.out.println(c%10); //2.25possiblein java
           }
}
```

```
public class OperatorDemo
 {
     public static void main(String args[])
      {
          int x=10;
          System.out.println(x++);    //10 (11)
          System.out.println(++x);   //12
          System.out.println(x--);    //12 (11)
          System.out.println(--x);    //10
      }
}
```

```
public class OperatorDemo
  {
       public static void main(String args[])
        {
            int a=10;
            int b=10;
           System.out.println(a++ + ++a);//10+12=22
           System.out.println(a); //12
           System.out.println(b++ + b++);//10+11=21
           System.out.println(b); //12

         }
  }
```

b) Relational Operators (Comparing two things)

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a Boolean value.

```
int a = 4;
int b = 1;
boolean c = a < b;
System.out.println("The C value is " + c) //false
System.out.println(a==4) //True
System.out.println(a!=b) //False
System.out.println(a<=4) //True
```

c) Boolean Logical Operators (Combining two or more relations)

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
?:	Ternary if-then-else

The outcome of these operations is a Boolean value.

Java logical Operators Example: & , &&

```
public class OperDemo
  {
     public static void main(String args[])
       {
          int a=10;
          int b=5;
          int c=20;
         System.out.println(a<b&&a<c);
         //false && true = false
         System.out.println(a<b&a<c);
        //false & true = false
       }
   }
```

Java logical Operators Example: & , &&

```
public class OperDemo
 {
        public static void main(String args[])
           {
             int a=10;
             int b=5;
             int c=20;
             System.out.println(a<b && a++<c);
                  //false && true = false
            System.out.println(a);
                 //10 because second condition is not checked
            System.out.println(a<b&a++<c);
               //false && true = false
            System.out.println(a);
              //11 because second condition is checked
           }
}
```

Java logical Operators Example: | , ||

```
class OperDemo
  {
     public static void main(String args[])
        {
            int a=10;
            int b=5;
            int c=20;
           System.out.println(a>b  || a<c);
            //true || true = true
           System.out.println(a>b | a<c);
            //true | true = true
           System.out.println(a>b || a++<c);
           //true || true = true
           System.out.println(a);
           //10 because second condition is not checked
           System.out.println(a>b | a++<c);
          //true | true = true
           System.out.println(a);
          //11 because second condition is checked
     }
}
```

Java logical Operators Example:

```
public class OperDemo
  {
        public static void main(String args[])
           {
             int a=10;
             int b=5;
             int c=20;
```

```
            System.out.println( a>b ^ a<c);
            //Both are same result is  - False
             System.out.println( a<b ^ a>c);
             // Both are same result is  - False
            System.out.println( a>b ^ a>c);
            // Both are different result is  - True
             System.out.println( a<b ^ a<c);
            // Both are Different result is  - True
        }
}

public class Operator
  {
      public static void main(String args[])
        {

          boolean c=true;
          boolean d=false;

          System.out.println(!c); //false
          System.out.println(!d); //true
        }
   }

public class OperatorExample
  {
      public static void main(String args[])
        {
              int a=20 , b=30;
              int c= (a>b) ? a  :  b;
              System.out.println("The big is " + c);
        }
   }
```

d) Bitwise Operators (Operates at bit level – on binary data)

Operator	Result
~	Bitwise unary NOT (one's complement)
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Java Unary Operator Example: ~ and !

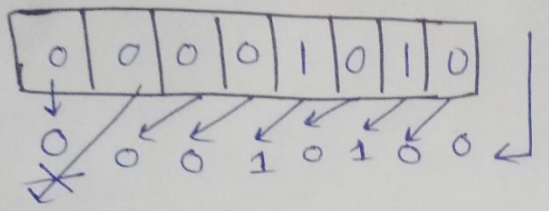
```
public class Operator
  {
      public static void main(String args[])
         {
            byte  a=10;
            byte b=-10;
            System.out.println(~a);  // -11 (one's Complement)
            System.out.println(~b); //    9  (one's Complement)

         }
   }
```

```
public class Operator
  {
      public static void main(String args[])
         {
            byte  a=10;
            byte b=8;
            byte c;
            c = a | b;
            System.out.println(c);
            c = a & b;
            System.out.println(c);
             c = a ^ b;
            System.out.println(c);

         }
   }
```

Java bitwise Operators Example: << , >>

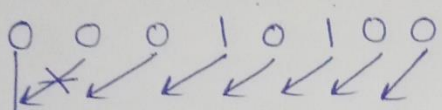
```
public class Oper

    {

       public static void main(String args[])

         {

                    byte a=10;
                    System.out.println(a<<2);  //40
                    byte b=-15;
                    System.out.println(b<<3); //-120
                    byte c=10;
                    System.out.println(c>>2);  //2
                    byte d=-15;
                    System.out.println(d>>1);  //-8
              }

}
```

1. Left shift (Doubles the value for each shift)



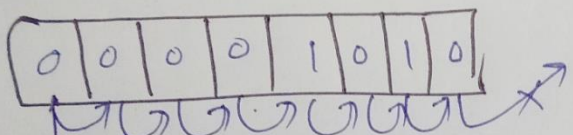
a << 1 ⇒ a = 10

1st time ⇒ a = 20

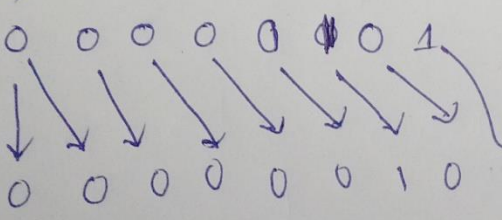


2nd time ⇒ a = 40

2. Right Shift (Halves the value for each shift)

① 

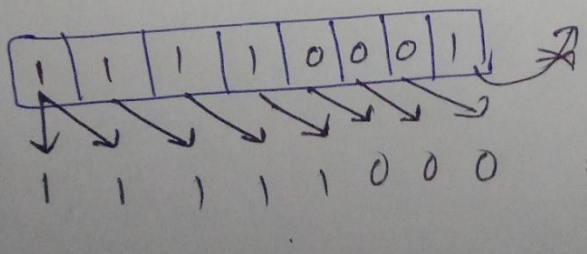
a >> 1 ⇒ a = 10



1st time ⇒ 5

2nd time ⇒ $\frac{5}{2}$ = ②

② a = -15



⇒

EuroKids Pre-School

Java bitwise Operators Example: >> , >>>

```
class Oper3
{
public static void main(String args[])
{
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
   System.out.println(20>>>2);

  //For nagative number, >>> changes MSB to 0
   System.out.println(-20>>2);
   System.out.println(-20>>>2);
  }
}

public class OperatorExample
   {
        public static void main(String args[])
         {
            int a=10;
            int b=20;
            a+=4;//a=a+4 (a=10+4)
            b-=4;//b=b-4 (b=20-4)
           System.out.println(a);
           System.out.println(b);
        }
}
```

Operator Precedence Table in Java

Operator	Category	Precedence
Unary	Postfix	*expr++* *expr--*
	Prefix	*++expr* *--expr* *+expr* *-expr* ~ !
Arithmetic	multiplicative	* / %
	Additive	+ -
Shift	Shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	Equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	Ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

```
public class OperatorExample
     {
       public static void main(String args[])
         {
            System.out.println(10*10/5+3-1*4/2);  //21
          }
       }
```

Java Assignment Operators Example

```
class OperatorExample
{
        public static void main(String[] args)
        {
                    int a=10;
                    a+=3;        //10+3
                    System.out.println(a);
                    a-=4;        //13-4
                    System.out.println(a);
                    a*=2;        //9*2
                    System.out.println(a);
                    a/=2;        //18/2
                    System.out.println(a);
                }
}
```

14. Arrays In Java

- An array is a group of like-typed (homogeneous) variables that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions. (One ,Two or More)
- A specific element in an array is accessed by its index. (index starts from 0)
- Arrays offer a convenient means of grouping related information.
- The memory for an array is dynamically allocated unlike C or C++.

1. One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables.
The general form of a one-dimensional array declaration is

```
type var-name[ ]; //declaration

int days[ ];
double avg[ ];
```

- Although this declaration an array variable, no array actually exists.
- The array is set to null, which represents an array with no value.
- To create physical array of integers, we must allocate using new operator.
- new is a special operator that allocates memory dynamically.

A one-dimensional array can be created as

```
array-var = new type [size]; //creation  or allocation
eg: days = new int[7];
    avg = new double[60];
```

Both declaration and allocation can be combined as

type var-name[ ] = new type [size]

```
int days[] = new int[7];
double avg[] = new double[60]
```

```
public class Array
{
public static void main(String args[])
  {
        int days[] = new int[7];
        days[0] = 12;  //assigning a value at index
        days[1] = 22;
        days[2] = 92;
        days[3] = 28;
        days[4] = 32;
        days[5] = 40;
        days[6] = 39;

       //Accessing  an array element
        System.out.println("The fourth element is "+ days[3]);

       //Updating an array element
        days[2]=days[2]+99;
        //Accessing  an array
        for(int i=0; i<6;i++)
        {
              System.out.println( days[i]);
        }

    }
}
```

Array Initialization

- Arrays can be initialized when they are declared.
- An array initializer is a list of comma-separated expressions
- Surrounded by curly braces.
- The commas separate the values of the array elements.

type arr_name[size] = { List of elements separated by comma}

Eg: int days[7] = {7,9,5,12,45,23,10};

- The size of array need not to be specified.
- The array will automatically be created large enough to hold the number of elements we specify in the array initializer.

Eg: int days[ ] = {7,9,5,12,45,23,10};

```
public class Average
{
   public static void main(String args[])
     {
                        double a[] = {75.1, 76.2, 65.3, 77.4, 84.5};
                        double sum = 0;
                        int i;
                        for(i=0; i<5; i++)
                                sum = sum + a[i];

                        System.out.println("Average is " + sum / 5);
                }
}
```

```
import java.io.*;
import java.util.*;
public class  ArrDemo
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the number of elements");
                 int n = sc.nextInt();

                 int a[] =  new int[n];

                 int sum =0;
                 System.out.println("Enter the  elements");
                     for(int i =0; i<n ; i++)
                          {
                                  a[i] = sc.nextInt();
                                  sum = sum+a[i];
                          }

                 System.out.println(" the  elements are");
                          for(int i =0; i<a.length ; i++)
                                    System.out.println(a[i]);

                  System.out.println("The sum of elements is " +sum);


        }
}
```

2. Multi-Dimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays.
To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Eg: A two-dimensional array can be declared and allocated as

```
int twoD[][] = new int[4][5];
```

as

```
class TwoD
{
public static void main(String args[])
        {
                int a[][]= new int[4][5];
                int i, j, k = 0;
                for(i=0; i<4; i++)
                for(j=0; j<5; j++)
                {
                        a[i][j] = k;
                        k++;
                }

                for(i=0; i<4; i++)
                {
                        for(j=0; j<5; j++)
                                System.out.print(a[i][j] + " ");
                        System.out.println();
                }
        }
}
```

- When you allocate memory for a multidimensional array we need to only specify the memory for the first (leftmost) dimension.
- We can allocate the remaining dimensions separately.

```
int a[][] = new int[4][ ];
    a[0] = new int[3];
    a[1] = new int[3;
    a[2] = new int[3];
    a[3] = new int[3];
```

- We can allocate each row with different number of elements; such an array is called **Jagged / Ragged Array. i.e. Each row can be of different columns.**

```
int a[][] = new int[4][ ];
a[0] = new int[2];
a[1] = new int[3];
a[2] = new int[4];
a[3] = new int[5];
```

- A multi dimensional array can be initialized as

```
int a[3][3] = {{1,2,3}, {3,4,5},{6,7,8}};
int a[3][] = {{1,2,3}, {3,4,5},{6,7,8}};
int a[4][] = {{1,2,3,4}, {3,4 },{6,7,8,9},{1,2}}; //Jagged array
```

```
import java.io.*;
import java.util.*;
//Jagged or Ragged Array – from keyboard
public class  ArrDemo
{
  public static void main(String[] args)
  {
            Scanner sc = new Scanner(System.in);
            System.out.println("Enter the number of rows");
              int m = sc.nextInt();
              int a[][] = new int[m][ ];
             int n;
              for(int i =0; i< m;i++)
              {
                 System.out.println("Enter the number of elements in " +i);
                 n = sc.nextInt();
                 a[i] = new int[n];
              }

           System.out.println("Enter the  elements");
            for(int i = 0; i<m ; i++)
                    for(int j=0;j<a[i].length;j++)
                              a[i][j] = sc.nextInt();

            System.out.println("the matrix elements are");
               for(int i = 0; i<m ; i++)
                           {
                                 for(int j=0;j<a[i].length;j++)
                                           System.out.print(" " +a[i][j]);
                                  System.out.println();
                             }
    }
}
```

```
//3. Matrix Multiplication
import java.util.*;
public class  MatrixDemo
{
        public static void main(String  args[])
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter the size of Matrix A  - m * n");
                        int m = sc.nextInt();
                        int n = sc.nextInt();

                System.out.println("Enter the size of Matrix B  - p * q");
                        int p = sc.nextInt();
                        int q = sc.nextInt();
                if (n != p)
                {
                System.out.println("Multiplication cannot be performed");
                        System.exit(0);
                }
                        int a[][] = new int[m][n];
                        int b[][] = new int[p][q];
                       int c[][] = new int[m][q];


            System.out.println("Enter the elements of Matrix-A");
                    for(int i =0; i<m;i++)
                        for(int j=0;j<n;j++)
                   a[i][j] = sc.nextInt();

             System.out.println("Enter the elements of Matrix-B");
                    for(int i =0; i<p;i++)
                        for(int j=0;j<q;j++)
                   b[i][j] = sc.nextInt();
```

```
        // Actual Code

         for(int i =0 ; i<m ;i++)
         {
               for(int j=0;j<q;j++)
                  {
                              c[i][j]=0;
                             for(int k=0;k<p;k++)
                                     c[i][j] = c[i][j] + a[i][k]*b[k][j];
                  }
         }

       System.out.println("The Product  Matrix-C  is");
                  for(int i =0; i<m;i++)
                        {
                            for(int j=0;j<q;j++)
                                    System.out.print("  " + c[i][j]);
                           System.out.println();
                        }

         }
}
```

Alternative Array Declaration Syntax

```
type[ ] var-name;
int a[]     = new int[3];
int[]   a1 = new int[3];

char a[][]     = new char[3][4];
char[][]    a  = new char[3][4];
```

********This alternative declaration form offers convenience when declaring several arrays at the same time.

```
int[]  a,b,c;          // create three arrays
char[][] a,b,c;
```

creates three array variables of type int. It is the same as writing

```
int a[], b[], c[]; // create three arrays
```

1. Type Conversions and Type Casting

When you assign value of one data type to another, the two types might not be compatible with each other.

If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion.

Eg: it is always possible to assign an int value to a long variable.

If the data types are not compatible then they need to be casted or converted explicitly.
Eg: there is no automatic conversion defined from double to byte.

Java is a strongly typed language

6. **Every variable has a type, every expression has a type, and every type is strictly defined.**

7. **All assignments, whether explicit or via parameter passing in method calls, are checked for Type compatibility.**

8. **There are no automatic coercions or conversions of conflicting types as in some languages.**

9. **The Java compiler checks all expressions and parameters to ensure that the types are compatible.**

10. **Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.**

Java Type conversion are in two forms

3. **Automatic Type Conversion / Widening / Type Coersion**

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place.

Eg: int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

For widening conversions

- The numeric types, including integer and floating-point types, are compatible with each other.

```
Eg: int a=10;
float f=a;
System.out.println(a);  //10
System.out.println(f);  //10.0
```

- No automatic conversions from the numeric types to char or boolean. (In compatible Types)

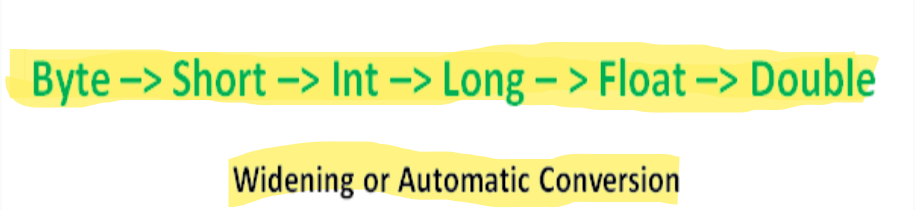
```
byte a =25;
boolean flag = a;
System.out.println("The value of a is " +a); //25
System.out.println("The value of flag is " +flag); //Error
```

```
// Incompatible types
```

- char and boolean are not compatible with each other. **(In compatible Types)**

```
boolean p = true;
char ch = p;
System.out.println("The value of p is " +p);  //true
System.out.println("The value of ch is " +ch); //Error
// Incompatible types
```

- Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.



4. Explicit Type Conversion / Narrowing / Type Casting

- The automatic type conversions will not fulfill all needs.
- For example an int value to a byte variable conversion will not be performed automatically, because a byte is smaller than an int. **(In compatible Types)**

```
int m = 67;
byte b = m;
System.out.println("The value of m is " +m); //67
System.out.println("The value of ch is " +b);
```

```
//Error- possible loss of precision
```

This kind of conversion is sometimes called a narrowing conversion, since we are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, we must use a cast.

A cast is simply an explicit type conversion.
It has this general form:

```
(target-type) value
```

```
int m = 67;
byte b = (byte)m;
System.out.println("The value of m is " +m);  //67
System.out.println("The value of b is " +b);  //67
```

- **When conversion from an int to a byte, If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.**

```
int m = 130;
byte b = (byte)m;
System.out.println("The value of m is " +m); //130
System.out.println("The value of b is " +b);  //-126
```

- **When a floating-point value is assigned to an integer type: truncation is occurred. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.**

```
double d = 234.56;
int i = (int)d;
System.out.println("The value of m is " +d); //234.56
```

```
System.out.println("The value of b is " +i); //234
```

- When a floating-point value is assigned to an integer type: if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

```
double d = 131.75;
byte i = (byte)d;
System.out.println("The value of m is " +d); //131.75
System.out.println("The value of b is " +i); //-125
```

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

```
// Demonstrate Type Casting.
class Type2
{
public static void main(String args[])
 {
        byte b;
        int i = 257;
        double d = 323.142;

                b = (byte) i;
        System.out.println("i and b " + i + "->" + b);

                i = (int) d;
        System.out.println("d and i " + d + "->" + i);

                b = (byte) d;
        System.out.println("d and b " + d + "-> " + b);
        }
}
```

Automatic Type Promotion in Expressions

1. In an expression , Java automatically promotes each byte, short, operand to int when evaluating an expression

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

2. The following code is an error.

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

//Because the expression is evaluated to int
Type conversion must be done explicitly as

```
byte b = 50;
b = (byte)(b * 2); //20
```

3.
```
class Simple
{
    public static void main(String[] args)
    {
    byte a=10;
    byte b=10;
    //byte c=a+b;//Compile Time Error: because a+b=20
    //will be int
    byte c=(byte)(a+b);
    System.out.println(c);  //20
    }
}
```

2. COMMAND LINE ARGUMENTS

- The java command-line argument is an argument passed at the time of running the java program to main method.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behavior of the program for the different values
- All command line arguments are of type String.

Simple example of command-line argument in java

```
public class Demo
        {
             public static void main(String args[])
              {
                 System.out.println("First argument is: "+args[0]);
              }
        }
```

compile by > javac Demo.java
run by > java Demo Vardhaman

Output: First argument is: Vardhaman

Example of command-line argument that prints all the values

```
class Demo{
public static void main(String args[])
        {
         for(int i=0;i<args.length;i++)
              System.out.println(args[i]);
```

```
        }
    }
```

```
// Performing Arithmetic Operations using command line arguments
class CommandDemo
{
        public static void main(String[] args)
        {
                int a = Integer.parseInt(args[1]);
                int b = Integer.parseInt(args[2]);
                int c=0;
                switch(Integer.parseInt(args[0]))
                {
                        case  1 : c=a+b;
                                  System.out.println("Add " + c);
                                    break;
                        case  2 : c=a-b;
                                   System.out.println("Sub " + c);
                                     break;
                        case 3 : c=a*b;
                                  System.out.println("Mul " + c);
                                   break;
                        case 4 :  c=a/b;
                                   System.out.println("Div" + c);
                                    break;
                        case 5 : c=a%b;
                                   System.out.println("Rem " + c);
                                     break;
                    }
              }
}
```

3.Classes, Objects and Methods

- The class is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- The class forms the basis for object-oriented programming in Java.
- Any concept we implement in a Java program must be encapsulated within a class.

A class is declared by use of the class keyword.
A simplified general form of a class definition is

```
modifier class  ClassName
{
        modifier  type instance-variable1;
        modifier  type instance-variable2;
            ------------------
        modifier type instance-variableN;
        modifier type methodname1(parameter-list)
          {
              // body of method
          }
         modifier type methodname2(parameter-list)
          {
              // body of method
          }
              ------------------
        modifier type methodnameN(parameter-list)
          {
              // body of method
         }
}
```

- **A class defines a new data type; using it we can create objects.**
- **A class is a template for an object, and an object is an instance of a class.**
- **When we define a class, declare its exact form and nature.**
- **A class contains data and the code that operates on that data.**
- **The data, or variables, defined within a class are called instance variables, because each instance of the class (that is, each object of the class) contains its own copy of these variables**
- **The code is contained within methods.**
- **The instance variables are acted upon and accessed by the methods defined for that class.**
- **The methods and variables defined within a class are called members of the class.**

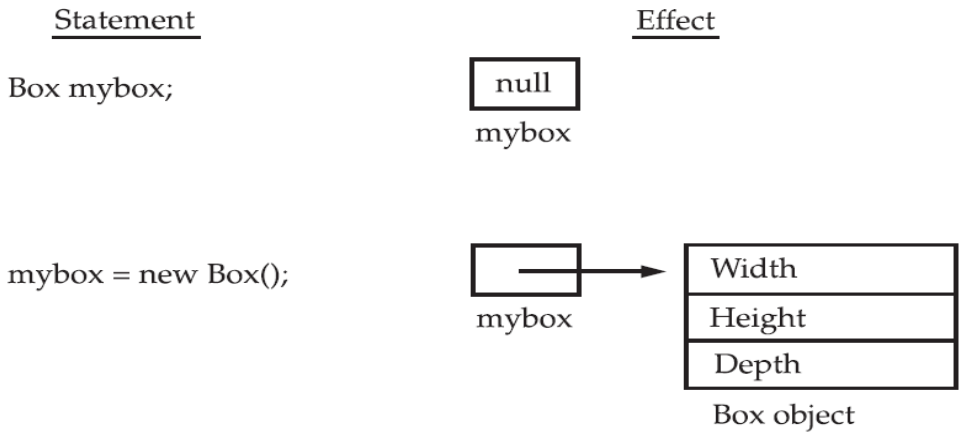
Example-1

```
class Box
 {
        double width;   // instance variable –heap memory
        double height;
        double depth;
        double volume()
          {
            double x;     // local variable – stack memory
             x= width * height * depth;
             return x;
            }
           void setDim(double w, double h, double d)
            {
                width = w;
                height = h;
                 depth = d;
            }
    }
class BoxDemo
 {
        public static void main(String args[])
          {
                Box b1= new Box();    //b1 – reference variable
                Box b2 = new Box();   //b2-reference variable
                b1.setDim(10, 20, 15);
                b2.setDim(3, 6, 9);
               double vol1 = b1.volume();
               System.out.println("box1 Volume is " +vol1);
               System.out.println("box1 Volume is " +b2.volume());
        }
}
```

Reference variables can be assigned

Eg1: Box mybox; //declaration

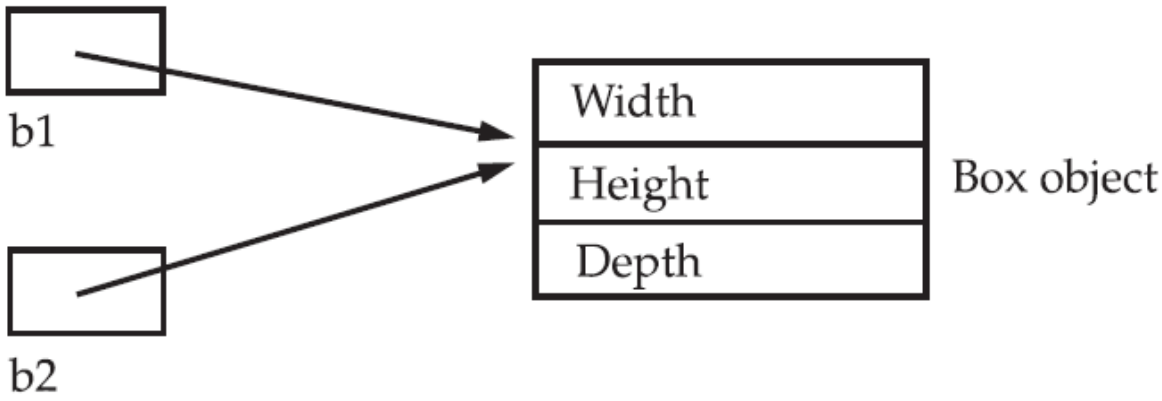
mybox=new Box(); //Allocation



Eg2 : Box b1 = new Box();

Box b2 = b1; // b1, b2 refers to same object

// if b1 = null; still b2 refers to same object.



Eg3 : Box b1 = new Box() , b2 = new Box(); // both are different

Eg4: Box b1 = new Box();

Box b2 = new Box(); // both are different

Example-2

```
class Employee
    {
        int id;
        String name;
        double salary;

       void insert(int i, String n, double s)
        {
           id=i;
           name=n;
           salary=s;
        }

      void display()
       {
         System.out.println(id+" "+name+" "+salary);
       }
}
public class TestEmployee
  {
        public static void main(String[] args)
         {
             Employee e1=new Employee();
             Employee e2=new Employee(),e3=new Employee();
             e1.insert(101,"Varun",45000);
             e2.insert(102,"Tharun",25000);
             e3.insert(103,"Nirun",55000);

 //new Employee().insert(104,”ram”,40000);  called  -anonymous object

              e1.display();
             e2.display();
              e3.display();
           }
    }
```

Example-3

```
class Account
  {
      int acc_no;
      String name;
      float amount;
      void create(int a,String n,float amt)
       {
          acc_no=a;
          name=n;
          amount=amt;
       }

       void deposit(float amt)
       {
           amount=amount+amt;
          System.out.println(amt+" deposited");
        }

      void withdraw(float amt)
       {
          if(amount<amt)
           {
                System.out.println("Insufficient Balance");
           }
          else
           {
                amount=amount-amt;
                System.out.println(amt+" withdrawn");
           }
       }

        void checkBalance()
          {
```

```
                System.out.println("Balance is: "+amount);
            }

        void display()
         {
            System.out.println(acc_no+" "+name+" "+amount);
         }
}

class TestAccount
  {
    public static void main(String[] args)
     {
       Account a1=new Account();
       a1.create(832345,"Ankit",1000);
       a1.display();
       a1.checkBalance();
       a1.deposit(40000);
       a1.checkBalance();
       a1.withdraw(15000);
       a1.checkBalance();
     }
}
```

4. Methods

The general form of a method:

```
type methodName(parameter-list)
 {
        // body of method
}
```

- **type specifies the type of data returned by the method.**
- **If the method does not return a value, its return type must be void.**
- **The name of the method is any valid identifier name.**
- **The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.**
- **If the method has no parameters, then the parameter list will be empty.**
- **Methods that have a return type return a value to the calling routine using**

  return value;

A method can be ->

1. **With No Parameters and No Return Value**

```
void volume()
  {
    double x;
     x= width * height * depth;
     System.out.println("The Volume is " +x);
    }
```

2. **With Parameters and No Return Value**

```
void volume(double len , double wid, double hgt)
  {
    double x=len*wid*hgt;
     System.out.println("The Volume is " +x);
    }
```

3. **With Parameters and a Return Value**

```
double volume(double len , double wid, double hgt)
  {
    double x= len*wid*hgt;
    return x;
    }
```

4. **With No Parameters and a Return Value**

```
double volume()
  {
    double x;
     x= width * height * depth;
     return x;
    }
```

Parameter Passing Techniques in Java

- **Formal Parameter:** A variable and its type as they appear in the prototype of the method.

  Syntax:

  function_name(datatype var1_name, datatype var2_name, ……)

```
double volume(double len , double wid, double hgt)  //Method Prototype
                {
                      double x= len*wid*hgt;
                      return x;
                 }
```

  - **len, wid and hgt are formal parameters**

- **Actual Parameter:** The variable or expression corresponding to a formal parameter that appears in the method call in the calling environment.

  Syntax:

  func_name(var1_name , var2_name , ……);

```
public class Demo
  {
     public static void main(String[] args)
       {
              double x=3.5;
              double x=4.5;
              double x=5.5;
              ---------------------
              double res = volume(x,y,z);  //Method Call
               ---------------------
              }
    }
```

  - **x,y,z are actual parameters**

1. Pass By Value:

- ✓ Changes made to formal parameter do not get transmitted back to the caller.
- ✓ Any modifications to the formal parameter variable inside the called method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.
- ✓ Inefficiency in storage allocation
- ✓ For objects and arrays, the copy semantics are costly

```
//Parameter Passing Program - Call By Value
    class Test
        {
          public static void update(int x, int y)
          {
             x++;
             y++;
             System.out.println("x,y values are" +x  + " , " +y);//11,21
          }
        }
     public class ByVal
         {
           public static void main(String[] args)
           {
              int a = 10;
              int b = 20;
              Test t = new Test();
          System.out.println("The values of a, b before method call");
          System.out.println("a, b values are" + a + "," + b); //10,20
                    t.update(a, b);
          System.out.println("The values of a, b after method call");
          System.out.println("a, b values are" + a + "," + b); //10,20
           }
      }
```

2. Call by reference(aliasing):

- ✓ Changes made to formal parameter do get transmitted back to the caller through parameter passing.
- ✓ Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference to the actual data.
- ✓ This method is efficient in both time and space.
- ✓ When we pass a reference, a new reference variable to the same object is created. So we can only change members of the object whose reference is passed. We cannot change the reference to refer to some other object as the received reference is a copy of the original reference.

```
class Demo
    {
      int a, b;
      void update(Demo obj)
      {
         obj.a += 10;
         obj.b += 20;
      }
    }
 public class ByRef
    {
      public static void main(String[] args)
      {
           Demo d = new Demo();
           d.a = 10;
           d.b = 20;
            System.out.println("a, b before method call");
            System.out.println( d.a + "," + d.b); //10,20
            d.update(d);  // Method Call
            System.out.println("a, b after method call");
            System.out.println("a, b values are" + d.a + "," + d.b); //20,40
       }
    }
```

5. Method Overloading

Polymorphism in Java is a concept by which we can perform a single action in different ways. i.e Poly means "many" and morphs means "forms".

The two type of Polymorphism

1. **Compile Time polymorphism**
2. **Run Time Polymorphism**

***Compile Time polymorphism in Java is achieved using "method overloading".**

Defining two or more methods within the same class that share the same name but parameter declarations are different. The methods are said to be overloaded and the process is referred as method overloading.

- **When an overloaded method is invoked uses type and number of arguments to determine which version is called.**
- **Overloaded methods must differ in type and/or number of parameters.**
- **When an overloaded method is called, Java simply executes the version of the method whose parameters match the arguments used in the call.**

Advantage of method overloading

1. **Method overloading *increases the readability of the program.***

***If we have to perform only one operation, having same name of the methods increases the readability of the program.**

2. **Provides Compile time Polymorphism**

Different ways to overload the method

There are two ways to overload the method in java

1. **By changing number of arguments**
2. **By changing the data type**

Method Overloading: By changing number of arguments

```
class Adder
 {
      int add(int a,int b)
        {
            return a+b;
        }
      int add(int a,int b,int c)
        {
            return a+b+c;
        }
   }

public class Test
  {
     public static void main(String[] args)
       {
          Adder a = new Adder();
         System.out.println(a.add(10,11)); //2-args Version
         System.out.println(a.add(10,11,12)); //3-args Version
       }
   }
```

***When a overloaded method is called, Java executes the version of the method whose parameters match the arguments used in the call.

2) Method Overloading: changing data type of arguments

```
class Adder
  {
       int add(int a, int b)
        {
           return a+b;
         }
       double add(double a, double b)
        {
            return a+b;
         }
   }

public class Test
     {
        public static void main(String[] args)
          {
            Adder a = new Adder();
            System.out.println(a.add(11,11));  //int version
            System.out.println(a.add(12.3,12.6));   //double version
         }
     }
```

***When a overloaded method is invoked, uses type and number of arguments to determine which version is called.

"Method Overloading is not possible by changing the return type of method only"

```
class Adder
  {
       int add(int a,int b)
         {
             return a+b;
         }
       double add(int a,int b)
         {
             return a+b;
         }
  }

public class Test
   {
        public static void main(String[] args)
          {
             Adder a = new Adder();
             System.out.println(a.add(11,11));//ambiguity

          }
    }
```

"We can overload java main () method"

Yes, by method overloading. We can have any number of main methods in a class by method overloading. But JVM calls main () method which receives string array as arguments only.

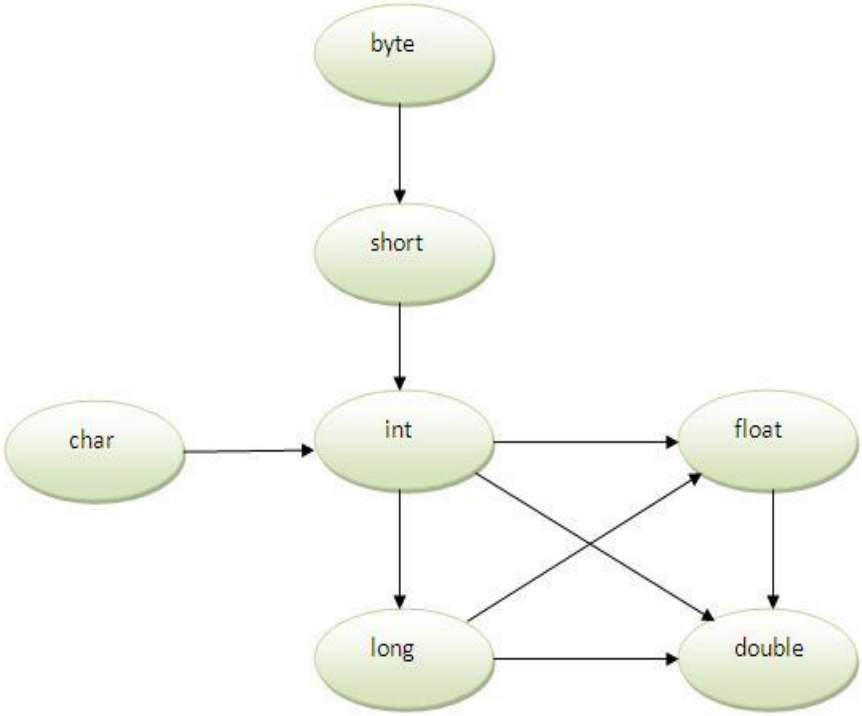
```
class Test
    {
      public static void main(String[] args)
       {
          System.out.println("main with String Array");
        }
      public static void main(String args)
         {
            System.out.println("main with String");
          }
       public static void main()
       {
         System.out.println("main without args");
       }
}
```

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching data type is found. (Widening Process).



- **As displayed in the above diagram, byte can be promoted to short, int, long, float or double.**
- **The short datatype can be promoted to int,long,float or double.**
- **The char datatype can be promoted to int,long,float or double and so on.**

Example of Method Overloading with Type Promotion

```
class Overloading
    {
       void sum(int a,long b)
        {
            System.out.println(a+b);
        }
      void sum(int a,int b,int c)
        {
            System.out.println(a+b+c);
        }

     public static void main(String args[])
      {
          Overloading obj=new Overloading();
          int a=20,b=20;
          obj.sum(a,b); //40
          //now second int literal will be promoted to long
          obj.sum(20,20,20);  //60
       }
}
```

***Example of Method Overloading with Type Promotion if matching found**

- If there are matching type arguments in the method, type promotion is not performed.

```
class Overloading
  {
      void sum(int a,int b)
        {
            System.out.println("int version" +(a+b));
         }
       void sum(long a,long b)
        {
            System.out.println("long version"+(a+b));
         }

 public static void main(String args[])
    {
          Overloading obj=new Overloading ();
           obj.sum(20,20); // int version executed
           long x=345, y=678;
           obj.sum(x,y); // long version executed

      }
}
```

Example of Method Overloading with Type Promotion in case of ambiguity

- If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class Overloading
   {
      void sum(int a,long b)
        {
            System.out.println(a+b);
        }
      void sum(long a,int b)
         {
             System.out.println(a+b);
         }
  public static void main(String args[])
     {
         Overloading obj=new Overloading ();
         obj.sum(20,20);//now ambiguity
     }
}
```

Output: Compile Time Error

//Example – Volume of various objects

```
class Volume
    {
        double PI= 3.142; //final to define constants
        double getVolume(double r)
            {
                return(4.0/3 * PI * r*r*r);
            }
        void getVolume(double r , double h)
            {
                System.out.print("The volume of Cylinder is ");
                double res = PI*r*r*h;
                System.out.println(res);
            }

        double getVolume(int r , int h)
            {
                return(1/3.0 * PI *r*r*h);
            }
        }

    public class VolumeDemo
    {
    public static void main(String args[])
        {
        Volume v1 = new Volume();
        System.out.println("The volume of Sphere is " + v1.getVolume(3.67));
        v1.getVolume(5.6,8.9);
        System.out.println("The volume of Cone is " +v1.getVolume(8,10));
        }
    }
```

//Example –area of various objects

```
class Shape
{
        public int area(int a)
        {
                System.out.print("Square Area is   ");
                return(a*a);
        }
        public double area(double r)
        {
                System.out.print("Area of Circle is   ");
                return(3.142 *r*r);
        }

        public void area(int a , int b)
        {
                System.out.println("The area of Rectangle is   " + (a*b));
        }

        public double area(double a , double b)
        {
          System.out.print("Area of Triangle is  ");
          return(0.5 * a* b);
        }
}

public class Test
{
         public static void main(String args[])
         {
                 Shape s = new Shape();
                 s.area(3,4); //Rectangle
                 System.out.println(s.area(5)); //Square
                 System.out.println(s.area(3.5)); //Circle
                 System.out.println(s.area(4.5,8.5)); //Triangle
         }
}
```

6. Constructors

- **Java Allows objects to be initialized themselves when they are created. i.e A constructor initializes an object immediately upon creation.**

- **Constructor in java is a special type of method that is used to initialize the object.**

- **Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object hence it is known as constructor.**

Characteristics

- **Constructor is a special type of method used to initialize the object.**
- **Once defined, constructor is automatically called after the object is created, before the new operator completes.**
- **Syntactically similar to a method**
- **By default constructor returns the object after constructing it.**
- **The name of the constructor must be same as its class name.**
- **If no constructor is defined, a default constructor is invoked by Java.**
- **Constructor must not have explicit return type. By default a constructor returns the class instance after creation.**

Rules for creating java constructor

There are basically two rules defined for the constructor.

1. **Constructor name must be same as its class name**
2. **Constructor must have no explicit return type**

Types of java constructors

There are two types of constructors:

1. Default constructor (No-Argument constructor)
   - ✓ Default constructor provides the default values to the object like 0 (int), 0.0(double), null (String) etc. depending on the type.
   - ✓ The Constructor has no arguments.
   - ✓ Initializes all objects to same, hence no longer used.

```
Box()
 {
   Len = wid= hgt =0.0;
 }
```

Eg: Box b1 = new Box();
Box b2 = new Box();

//Example for default constructor-1

```
public class Bike
 {
     Bike()
     {
       System.out.println("Bike is created");
      }
     public static void main(String args[])
      {
               Bike b=new Bike ();
       }
   }
```

```
// Example for default constructor-2

class Box
 {
    double len ;
    double wid;
    double hgt;

    Box()
    {
            len = wid= hgt =0.0;
    }
   double volume( )
    {
            double res;
            res = len * wid * hgt;
            return res;
    }
    void display()
    {
       System.out.println("The Dimensions of box are");
 System.out.println("Length= "+len +"Width=" +wid + "Height=" +hgt);
    }
}

public class const1
        {
    public static void main(String args[])
    {
            Box b1 = new Box();
            b1.display();
      System.out.println("The volume of b1 is " + b1.volume());
            Box b2 = new Box();
            b2.display();
      System.out.println("The volume of b2 is " + b2.volume());
    }
}
```

2. Parameterized constructor

- ✓ A constructor that has parameters is known as parameterized constructor.
- ✓ Used to create objects of different state and type.
- ✓ Parameterized constructor is used to provide different values to the distinct objects.

```
Box(double x)
    {
      len = wid= hgt =x;
    }

Box(double x ,double y , double z)
    {
        len = x;
        wid = y;
        hgt = z;
    }
```

```
//Example for Parameterized constructor-1
public class Student
 {
    int id;
    String name;
    Student (int i,String n)
    {
      id = i;
     name = n;
     }
   void display()
    {
      System.out.println(id+" "+name);
    }
  public static void main(String args[])
   {
     Student s1 = new Student (111,"Ramesh");
     Student s2 = new Student (222,"Suresh");
     s1.display();          s2.display();
   }
}
```

```
//Example for Parameterized constructor-2
class Box
 {
    double len ;
    double wid;
    double hgt;
    Box(double x ,double y , double z)
    {
            len = x;
            wid = y;
            hgt = z;
    }
   double volume( )
    {
            double res;
            res = len * wid * hgt;
            return res;
    }

    void display()
    {
            System.out.println("The Dimensions of box are");
             System.out.println(len +":" +wid + ":" +hgt);
    }
}
public class const1
{
    public static void main(String args[])
    {
             Box b1 = new Box(4,5,6);
             b1.display();
      System.out.println("The volume of b1 is " + b1.volume());
             Box b2 = new Box(7.5,8.5,12.5);
             b2.display();
      System.out.println("The volume of b2 is " + b2.volume());
    }
}
```

Copy constructor

- ✓ There is no copy constructor specific in java. But, we can copy the values of one object to another.
- ✓ Copy one object into another
  - By constructor
  - By assigning the values of one object into another

// Example for Parameterized constructor-1

//By constructor

```
public class Student
 {
   int id;
   String name;
   Student(int i,String n)
     {
         id = i;
         name = n;
      }

  Student(Student s)
     {
         id = s.id;
         name =s.name;
     }

   void display()
     {
         System.out.println(id+" "+name);
      }
```

```
    public static void main(String args[])
     {
        Student s1 = new Student(1001, "Varun");
        Student s2 = new Student(s1);
        s1.display();
        s2.display();
      }
}
```

//Copying values without constructor

- We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
public class Student
 {
      int id;
      String name;
      Student(int i,String n)
       {
          id = i;
          name = n;
       }
       Student( )
       {
        }
     void display()
       {
            System.out.println(id+" "+name);
       }
```

```
        public static void main(String args[])
         {
           Student s1 = new Student(121,"Tarun");
           Student s2 = new Student( );
           s2.id=s1.id;
           s2.name=s1.name;
                 s1.display();
                s2.display();
     }
   }
```

```
//Example for Copy constructor-2
  class Box
  {
     double len ;
     double wid;
     double hgt;
     Box(double x ,double y , double z)
     {
            len = x;
            wid = y;
            hgt = z;
     }
     Box(Box b)
     {
            len = b.len;
            wid = b.wid;
            hgt = b.hgt;
     }
    double volume( )
     {
            double res;
            res = len * wid * hgt;
            return res;
     }
```

```
    void display()
    {
            System.out.println("The Dimensions of box are");
             System.out.println(len +":" +wid + ":" +hgt);
    }
}

public class const1
{
    public static void main(String args[])
    {
             Box b1 = new Box(4,5,6);
             b1.display();
      System.out.println("The volume of b1 is " + b1.volume());
             Box b2 = new Box(b1);
             b2.display();
      System.out.println("The volume of b2 is " + b2.volume());

            Box b3 = new Box();
            b3.len = b2.len;
           b3.wid = b2.wid;
           b3.hgt = b2.hgt;

    }
}
```

Difference between constructor and method in java

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if we don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name is not same as class name

Does constructor return any value?

Yes, that is current class instance. (We cannot use return type explicitly, it returns an object by constructing it).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, opening a file, calling method etc. We can perform any operation in the constructor like we perform in the method.

```
Ex: Box()
    {
      System.out.println("It is a Default Constructor");
      int x = 22;
      x=x-2;
      System.out.println("The is  " +x);
     }
```

7. Overloading Constructors

- **Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter lists.**
- **The compiler differentiates these constructors by taking into account the number of parameters in the list and their data type.**
- **Like methods, the constructors in java can also be overloaded. This process is called constructor overloading.**

//Example of Constructor Overloading -1

```
public class Student
  {
        int id;
       String name;
       int age;
            Student()
              { }

           Student (int i,String n)
           {
              id = i;
             name = n;
           }
      Student (int i,String n,int a)
        {
            id = i;
           name = n;
           age=a;
         }
```

```
        void display()
          {

          System.out.println(id+" "+name+" "+age);

          }

        public static void main(String args[])
          {
         Student s1 = new Student(111,"Karan");
         Student s2 = new Student(222,"Aryan",25);
         Student s3 = new Student( ) ;
               s3.id=  333
               s3.name="Rajan";
               s3.age = 30;
                s1.display();
                s2.display();
                s3.display();
       }
}

//Example of Constructor Overloading -2
class Box
{
       double len ;
       double wid;
       double hgt;
        Box()
        {
               len =12;
               wid =13;
                hgt=15;
         }
```

```
        Box(double x)
         {
                len=wid=hgt=x;
          }

        Box(double x , double y , double z)
         {
                len =x ;
                wid =y;
                hgt = z;
         }

        Box(Box b)
        {
                len = b.len;
                wid = b.wid;
                hgt = b.hgt;
        }

        double volume( )
        {
                double res;
                res = len * wid * hgt;
                return res;
        }

        void display()
        {
                System.out.println("The Dimensions of box are");
                System.out.println(len +":" +wid + ":" +hgt);
        }
}
```

```
public class ConstOverloading
{
        public static void main(String args[])
        {
                Box b1 = new Box(); //default constructor
                b1.display();
         System.out.println("The volume of b1 is " + b1.volume());

                 Box b2 = new Box(3.5);
        //parameterized constructor -one parameter
                b2.display();
         System.out.println("The volume of b2 is " + b2.volume());

                Box b3 = new Box(3.7,4.6,5.8);
        //parameterized constructor - three parameter
                b3.display();
         System.out.println("The volume of b1 is " + b3.volume());

                Box b4 = new Box(b2); //copy constructor
                b4.display();
         System.out.println("The volume of b4 is " + b4.volume());

              //  Box b5 = new Box(3.4 , 5.5); //Error


        }
}
```

//Example of Constructor Overloading -3

```
class Employee
 {
        class Employee
        {
                int eid ;
                String name;
                double sal;

                Employee()
                {
                        eid=0;
                        name=null;
                        sal=0.0;
                }

                Employee(int x , String y , double z)
                {
                        eid = x;
                        name=y;
                        sal=z;
                }

                Employee(Employee e)
                {
                        eid = e.eid;
                        name = e.name;
                        sal = e.sal;
                }
```

```
 void display()
        {
          System.out.println("Employee ID is " + eid);
          System.out.println("Employee Name is " + name);
          System.out.println("Employee Salary is " + sal);
        }
}

public class emp
{
        public static void main(String args[])
        {
                Employee e1 = new Employee();
 Employee e2 = new Employee(1010 , "Rohith" , 75000);
                Employee e3 = new Employee(e2);
                e1.display();
                e2.display();
                e3.display();
        }
}
```

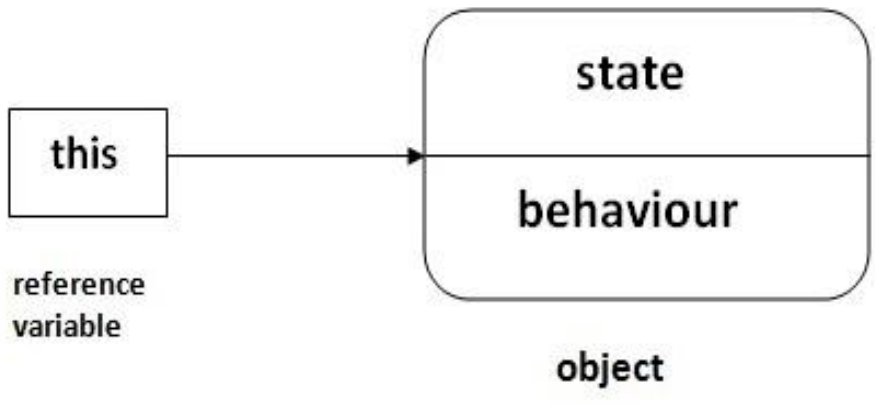
8. this keyword

In java, this is a reference variable that refers to the current object.

Usage of this keyword

1. **this can be used to refer current class instance variable.**
2. **this can be used to invoke current class method (implicitly)**
3. **this() can be used to invoke current class constructor.**
4. **this can be passed as an argument in the method call.**

- **Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.**
- **this can be used inside any method to refer to the current object.**



1) this: to refer current class instance variable

When a local variable has the same name as an instance variable, the local variable hides the instance variable. This is called instance variable hiding.(Naming Collision).

"this" refer directly to the object, we can use it to resolve any namespace collisions that might occur between instance variables and local variables.

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

```
class Student
 {
       int rollno;
       String name;
       float fee;
       Student(int rollno,String name,float fee)
        {
           rollno=rollno;
           name=name;
           fee=fee;
        }
      void display()
        {
          System.out.println(rollno+" "+name+" "+fee);
        }
   }
```

```
class TestThis
 {
        public static void main(String args[])
          {
            Student s1=new Student(111,"ankit",5000f);
            Student s2=new Student(112,"sumit",6000f);
             s1.display();
             s2.display();
          }
     }
```

Output: 0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
{
this.rollno=rollno;    //resolve the problem of instance variable hiding
this.name=name;
this.fee=fee;
}
void display()
 {
   System.out.println(rollno+" "+name+" "+fee);
  }
}
```

```
public class Test
{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```

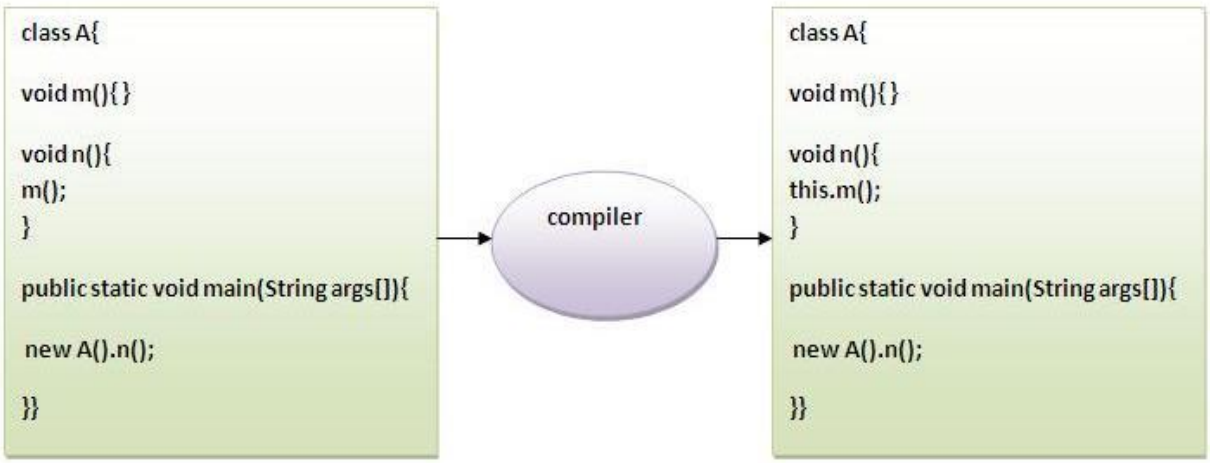
Output:

```
111 ankit 5000
112 sumit 6000
```

******It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.***

2) this: to invoke current class method

We may invoke the method of the current class by using this keyword. If we don't use the keyword, compiler automatically adds this keyword while invoking the method.



```
class A
  {
      void m( )
        {
           System.out.println("hello m");
        }
      void n( )
       {
          System.out.println("hello n");
         //m();//same as this.m()
          this.m();
       }
  }

 public class Test
     {
        public static void main(String args[])
          {
             A a=new A();
             a.n();
           }
      }
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
      class Student
        {
          int rollno;
          String name,course;
          float fee;
```

```
        Student(int rollno,String name,String course)
         {
           this.rollno=rollno;
           this.name=name;
           this.course=course;
         }
    Student(int rollno,String name,String course,float fee)
     {
        this(rollno,name,course);//reusing constructor
        this.fee=fee;
     }
    void display()
     {
        System.out.println(rollno+" "+name+" "+course+" "+fee);
     }
}

public class Test
  {
      public static void main(String args[])
        {
          Student s1=new Student(111,"ankit","java");
          Student s2=new Student(112,"sumit","java",6000f);
          s1.display();
          s2.display();
        }
  }
```

Call to this( ) constructor must be the first statement inside a constructor.

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2
  {
    void m(S2 obj)
     {
       System.out.println("method is invoked");
     }
    void p()
     {
       m(this);
     }
    public static void main(String args[])
     {
       S2 s1 = new S2();
       s1.p();
      }
  }
```

Output:

method is invoked

9. static keyword

- The keyword static is used in java for a variable, a block or a method.
- The static keyword belongs to a class but not to an object.
- static keyword is used with variable, method and block.

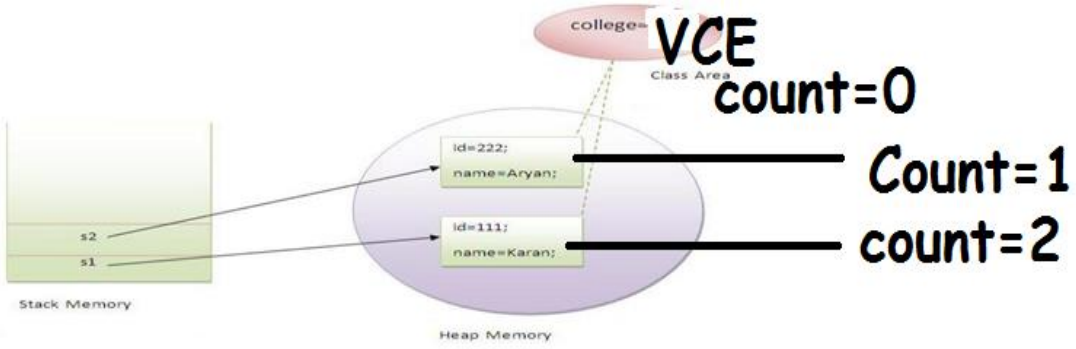
1. Java static variables.

- Static keyword is used to define a class member that will be used independently of any object of that class. Also called class variables
- A variable declared as static, then it is called static variable.
- Instance variables declared as static are global variables.
- When an object of such class is created no copy of the static variable will exist. All instances of class share same variables.
- A static variable can be used to refer to common property of all objects.

  Eg: Company Name, College Name etc.
- The static variables gets memory only once in a class area at the time of class is loading in to memory.
- Static variables make the program memory efficient. i.e. it saves memory.
- A static variable can be accessed using class name without instantiating it.

  Eg: Student.collegename

```
//static instance variables
        class Student
            {
                int rollno;
                String name;
                static String college ="VCE";
                static int count=0;
                    Student(int r,String n)
                     {
                        rollno = r;
                        name = n;
                         count++;
                    }
                  void display ()
                  {
                       System.out.println(rollno+" "+name+" "+college);
                              System.out.println("count "+count);
                   }
                }
          public class StaticDemo1
                 {
                      public static void main(String args[])
                          {
                                Student s1 = new Student(111,"Karan");
                                s1.display();
                                Student s2 = new Student(123,"Aryan");
                                 s2.display();
                                Student s3 = new Student(546,"Rajan");
                                s3.display();
System.out.println("The number of students " +Student.count); //3
                                }
                }
```




```
class Counter
    {
       static int count=0;
       //will get memory only once and retain its value
      Counter ()
        {
            count++;
            System.out.println(count);
         }
      public static void main(String args[])
        {
           Counter c1=new Counter ();
           Counter c2=new Counter ();
           Counter c3=new Counter ();
        }
   }
```

Output:

```
1
2
3
```

2. Java static block

- A block of code in a class, used to initialize the static data member.
- The static block is executed before main() method at the time of class is loading.
- A class can have any number of static blocks, executes them in the order they defined.
- A static block can access only static data.

```
//Example-1 for static block
public class  StatDemo
{
        int a;
        static int b;
         static    {
                      // a=10; //CTE
                      b=0;
                      b=b+12;
                      System.out.println("Static block 1 - " +b);
                    }

           static {

                      b=b-5;
                      System.out.println("Static block 2 - " +b);
                   }
        public static void main(String[] args)
        {
        System.out.println("Hello World!- MainMethod started");
        }
}
```

//Example-2 for static block

```
public class  StaticDemo
{
   static int x ;
    static {
                System.out.println("First Static Block");
                x =10;
                System.out.println("x is " +x);
            }

    static {
                 System.out.println("Second Static Block ");
                 x = x+20;
                 System.out.println("x is "+x);
            }


    public static void main(String[] args)
    {
            System.out.println("Hello World!- ");
    }
}
```

3. Java static method

- If we apply static keyword with any method, it is static method.
- The restrictions of static on methods declared is
  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to with "this" or "super" keys in any way. Because the methods belongs to a class but not to objects.
- A static method belongs to a class rather than object.
- A static method can be invoked without the need for creating an instance of a class. (object)
- Cannot access non static data variables, data or methods directly.
- A static method is used to initialize or modify the static data.
- A static method is can be accessed without creating an instance of a class (object) directly using ClassName.

  Eg: ClassName.methodName();

Why Java main() method is static?

Java main method is static because

i) An object is not required to call static method.

ii) If it is non static method, JVM creates object first then call main( ) this leads to problem of extra memory allocation.

```
//static method example -1

class Test
        {
         static   int cube(int x)
           {
            return x*x*x;
           }
         static void display()
          {

             int x ;

             x = cube(6);

             System.out.println(x);
           }


         }

public class StaticDemo2
{
         public static void main(String args[])
           {
              int result=Test.cube(5);  //Directly using class name
              System.out.println(result);
              Test t1 = new Test();
               t1.display();
           }
}
```

```
//static method example -2

public class StaticDemo3
{
    int a=40;//non static
    static int b=20;
    public void show()
    {
        System.out.println("Non static method");
    }
    public static void display()
    {
        b = b+5;
        System.out.println("Static Method");
        System.out.println(b);
        // a=a-5; //CTE
    }
    public static void main(String args[])
    {
        //  a=a+5; //CTE
        System.out.println(a);
        b=b+45;
        System.out.println(b);
        display();
        // show();  // CTE
    }
}
```

//static method example -2

```
class Calculate
      {
         static int cube(int x)
          {
           return x*x*x;
           }

         public static void main(String args[])
          {
             int result=Calculate.cube(5);
             System.out.println(result);
           }
      }


        class A
         {
            int a=40;//non static
            public static void main(String args[])
              {
                   System.out.println(a);
               }
         }
```

Output: Compile Time Error

//Example-6

1.String Class

- String is a sequence of characters.
- But in Java, string is an object that represents a sequence of characters.
- The java.lang.String class is used to create a string object.
- Java String class provides methods to perform operations on string

- A String object is created using
  - By String Literal

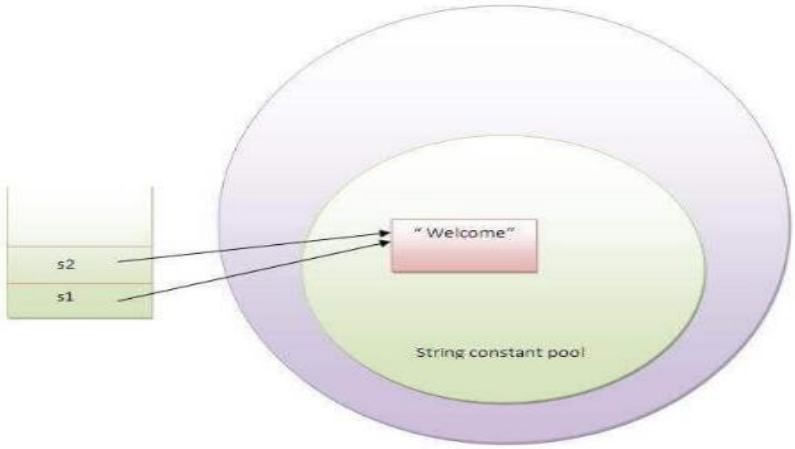
Java String literal is created by using double quotes.

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

```
String s1="Welcome";
String s2="Welcome"; //It doesn't create a new instance
```



//Example-6

- **By new keyword.**

String class supports several constructors to create strings.

1. **String s = new String( );**
   //default constructor – empty string

2. String s=**new** String("Welcome");

3. ***String(char chars[ ]);***
   // parameterized constructor - char array to string object
   char ch[] = { 'w', 'e', 'l','c','o','m','e' };
   String s1 = new String(ch); //"welcome"

4. **String(char *chars*[ ], int *startIndex*, int *numChars*)**
   // parameterized constructor – char array to string object
   char chars[] = { 'w', 'e', 'l','c','o','m','e' };
   String s2 = new String(chars, 3,4); //"come"

5. **String(byte *asciiChars*[ ])**
   String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

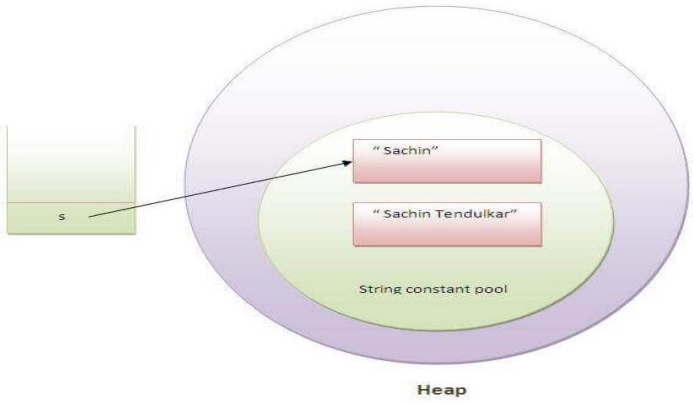
   byte ascii[] = {65, 66, 67, 68, 69, 70 };
   String s1 = new String(ascii); //ABCDEF
   System.out.println(s1); //"ABCDEF"
   String s2= new String(ascii , 2,3); //"CDE"

6. **String(String *strObj*) // copy constructor**
   String s1 = new String("Hyderabad");
   String s3 = new String(s1);

//Example-6

- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.
- Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created with changes.

```
class String1
 {
 public static void main(String args[])
  {
    String s="Sachin";
    s.concat(" Tendulkar");
    System.out.println(s); // Sachin
  }
}
```



- Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why string is known as immutable.
- As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".
- But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

```
//Example-6

class String1
 {
  public static void main(String args[])
   {
     String s="Sachin";
     s= s.concat(" Tendulkar");
     System.out.println(s);   // SachinTendulkar
   }
}
```

Reason: Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin". If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

- For mutable strings, we can use **StringBuffer** and **StringBuilder** classes are used.

The methods of String Class are

Method Name	Description	Example

int length()	returns string length
	
	public int length()
		String s1="java";
		System.out.println("string length is: "+s1.length()); //4
		
		System.out.println("string length is: "+"java".length()); //4

char charAt(int index)	Extracts a single character from the string. Returns char value for the particular index. public char charAt(int index)	String str="welcome"; char ch=str.charAt(3); //c

//Example-6

String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
 	
Used to extract a substring from specified index from the main string.
Returns a substring.
		String s1="welcome";
		String str1= s1.substring(3); //come
		String str2 = s1.substring(3,5); //com

String toLowerCase()	Returns a string in lowercase. public String toLowerCase()	String s1="OBJECT oreinTED"; String str=s1.toLowerCase(); System.out.println(str); //object oreinted
String toUpperCase()	Returns a string in uppercase. public String toUpperCase()	String s1="OBJECT oreinTED"; String str=s1.toUpperCase(); System.out.println(str); //OBJECT ORIENTED
void getChars (int start, int end, char target[] , int tarstart) Used to extract more than one character at a time.		Char buf[ ] = new char[10]; String str= "Welcome to Hyderabad"; str.getChars(3,6,buf,0) //come
byte[] getBytes()	Stores the characters in an array of bytes.	Strng str = new String("Welcome to India"); byte b[ ] = str.getBytes();
char[] toCharArray()	Converts all the characters of A String object into a character array.	Strng str = new String("Welcome to India"); char ch[] = str.toCharArray();
String Comparison boolean equals(Object str)	Returns true if the strings contain same characters in same order	String s1="hello"; String s2="hello"; String s3="HELLO"; System.out.println(s1.equals(s2));

//Example-6

	otherwise returns false.	//true
		System.out.println(s1.equals(s3));
		//false

boolean equalsIgnoreCase(String str)	Compares two strings for equality and ignores the case differences.	boolean x=s1.equalsIgnoreCase(s3)); //true

boolean startsWith()	Returns true, if the given string begins with a specified string.	System.out.println("welcome".startsWith("wel")); //true
		String str = new String("Welcome");
		System.out.println(str.startsWith("come")); //false

boolean endsWith()	Returns true, if the given string ends with a specified string.	System.out.println("welcome".endsWith("wel")); //false
		String str = new String("Welcome");
		System.out.println(str.endsWith("come")); //true

equals() versus ==	equals() -> Compares the characters inside a string object.
	== ->Compares two object references to see whether they refer to same instance.
		String s1 = new "welcome";
		String s2 = new String(s1);
		String s3= s1;
		System.out.println(s1.equals(s2));
		//true
		System.out.println(s2==s3) ; //false
		System.out.println(s1==s3) ; //true

int compareTo(String str)		

//Example-6

int compareToIgnoreCase(String str)	Used to compare to strings.
	It returns
	< 0: The invoking string is less than the given string.
	>0: The invoking string is greater than the given string.
	0: if the two strings are equal.
	
	Same result as compareTo().
	But ignores the case differences.
		String s1 =”welcome”;
		String s2= “hello”;
		String s3 = “welcome”;
		System.out.println(s1.compareTo(s3)); //0
		System.out.println(s1.compareTo(s2)); // >0
		System.out.println(s2.compareTo(s3)); // <0
		
		String s1 =”welcome”;
		String s3 = “WELcome”;
		int x=s1.compareToIgnoreCase(s3));
		//0

Searching Strings int indexOf(int ch) int indexOf(String str) int indexOf(int ch , int start) int indexOf(String str , int start) int lastIndexOf(int ch) int lastIndexOf(String str)	Used to search a string for a specified character or substring. Returns index at which the character or string is found , Otherwise returns -1.	String str = “to do welcome to indian railways to come and well do of the rail works” int x = str.indexOf(‘w’); int x = str.indexOf(“wel”); int x = str.indexOf(‘w’ , 10); int x = str.lastIndexOf(‘r’);
Modifying Strings String substring(int start)	To create a new copy of modified	String s1="welcome";

//Example-6

	string the functions are used.	String str1= s1.substring(3); //come
String substring(int start , int end)	Returns a substring from specified index.	String str2 = s1.substring(3,5); //com
String concat(String str)	The two strings are concatenated.	String s1 = "Welcome"; String s2 = s1.concat("To India");
String replace(char old , char new)	Replaces all occurrences of one character in the invoking string with another character.	String s1 = "Welcome"; String s2 = s1.replace('e' , 'o');
String trim()	Returns a copy of invoking string from which any leading and trailing white space has been removed.	String s1 = " Welcome "; String s2 = s1.trim(); //"welcome"

+	String concatenation	// String Concatenation
		int year=2019;
		String s2 = s1 + " " + year;
		System.out.println(s2);
		System.out.println(s2+2+2);
		System.out.println(s2+(2+2));

String toString()	Returns a String that contains the human readable	String str="Welcome to Hyd"; String s = str.toString();

//Example-6

	form, which describes an object of a class.	Box b1 = new Box(); String str = b1.toString();
boolean isEmpty()	Returns true if the string object is empty.	String s=""; boolean x = s.isEmpty(); //true
String[] split(String exp)	Returns a split string matching regular expression.	String str="Welcome to Hyd for biryani"; String s[] = str.split("\\s"); Or String s[] = str.split(" "); //splits string with white spaces.

```
//Converting String to particular type
public class  Demo
{
        public static void main(String[] args)
        {
                String str1= "27";
                //convert string to int
                 int x = Integer.parseInt(str1);
                String str2= "345.67";
                  //convert string to double
                 double d = Double.parseDouble(str2);
                String str3="34.56";
                   //convert string to float
                 float f = Float.parseFloat(str3);
                  String str4="4555526522426";
                //convert string to long
                  long y= Long.parseLong(str4);
```

```
        //Example-6

                                double res=  (d*y+x) / f;
                                System.out.println("The result is " +res);
                        }
                }
        //String Operations

                import java.io.*;

                public class  StringOp1
                {
                        public static void main(String[] args)
                        {
                                // String length
                               String s1 = "Welcome to Java String Methods";
        System.out.println("The no of characters are " + s1.length());

                              // String Concatenation
                               int year = 2019;
                               String s2 = s1 +  "  " + year;
                              System.out.println(s2);
                              System.out.println(s2+2+2);
                              System.out.println(s2+(2+2));

                              String str1= new String("Welcome");
                              String s = str1.concat(" to India");
                              System.out.println(s);

                             // Character Extraction
                             //String s1 = "Welcome to Java String Methods";

                              char ch1 = s1.charAt(3);
                              System.out.println(ch1);
```

```
//Example-6

        char ch2[] = new char[5];
        s1.getChars(5,10,ch2,0);
        System.out.println(ch2);

        char ch3[ ] = new char[s1.length()];
        ch3 = s1.toCharArray();
        System.out.println(ch3);

    // String Comparisons

        String s3 = new String("WELCOME");
        String s4 = "WELCOME";
        String s5 = "welcome";
        System.out.println(s3.equals(s4)); //true
        System.out.println(s3.equals(s5));  //false
    System.out.println(s3.equalsIgnoreCase(s5)); //true
            String s6 = s3;
        System.out.println(s3==s4); //false
        System.out.println(s3==s6); //true

        String s7 = "Food Time";

    System.out.println(s7.startsWith("Food")); //true
    System.out.println(s7.endsWith("Time")); //true
    System.out.println(s3.compareTo(s5)); //-ve value

  //Searching Strings

System.out.println("The first occurance of l  is " +
s1.indexOf('l'));
```

```
//Example-6

 System.out.println("The last occurance of l  is " +
s1.lastIndexOf('l'));

System.out.println("The first occurance of l  is from 5 " +
s1.indexOf('l',5));
 System.out.println("The last occurance of l  is from 5 " +
s1.lastIndexOf('l',5));
        System.out.println("The first occurance of me  is
        from 5 " + s1.indexOf("me",7))

        System.out.println("The last occurance of me  is from
        5 " + s1.lastIndexOf("me",7));

         //Modifying a string
  String str11 =  s1.substring(5);
  String str22 = s1.substring(5, 15);
  System.out.println(str11);
  System.out.println(str22);
   String str3 = s1.concat("Strings");
   System.out.println(str3);
   String str4 =s1.replace("Java" , "Advanced Java");
   System.out.println(str4);

    String str5 = "      Welcome       ";
    System.out.println(str5);
    System.out.println(str5.trim());

    System.out.println(str5.toLowerCase());
    System.out.println(str5.toUpperCase());

    String s11 ="";
    System.out.println(s11.isEmpty());
```

```
        //Example-6

                        String s22 = "welcome to India with love";
                        String ss[ ]  = s22.split(" ");
                         for(int i=0; i< ss.length; i++)
                              System.out.println(ss[i]);

                }
        }
//Program to use split() (Even Words list)
import java.util.*;
public class Task
{
        public static void main(String args[])
        {
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a line of Text");
                String line = sc.nextLine();

                String s[]= line.split(" ");
                int count=0;
                //for each loop
                for(String word : s)
                {
                        int len = word.length();
                        if(len % 2 == 0)
                        {
                                count++;
                        System.out.println(word + "    : " + len + "   : " +count);
                        }
                }
        }
}
```